Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Koncepty v C++

Bakalárska práca

2024
Marek Lošonský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Koncepty v C++

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. RNDr. Robert Lukoťka, PhD.

Bratislava, 2024
Marek Lošonský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Marek Lošonský
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Koncepty v C++
Concepts in C++

Anotácia: Koncepty v C++ sú jazykový konštrukt vyjadrujúci predikáty počítajúce sa v čase kompilácie. Koncepty sú základným stavebným prvkom generického programovania, hrajú v ňom podobnú úlohu ako interfacy v objektovo orientovanom programovaní. Napriek tomu, že koncepty ako pojem sa v rámci dokumentácie STL používajú od 90-tich rokov, jazyková podpora konceptov sa do štandardu dostala iba nedávno (C++20). Koncepty predstavujú z mnohých ohľadov zaujímavú výzvu pre tvorcov softvéru. V porovnaní s interfacami umožňujú oveľa väčšiu flexibilitu. Okrem toho je obvyklé, že súčasťou konceptu sú aj predpoklady, ktoré nemožno zachytiť existujúcimi prvkami jazyka (mimo komentárov). Cieľom bakalárskej práce je poskytnúť prehľad aktuálneho stavu problematiky. Prehľad by mal zahŕňať niektoré z nasledujúcich tém: koncepty v C++ štandarde, koncepty v kompilátoroch, najlepšie postupy pri použití konceptov v aplikačnom a knižničnom kóde, návrhy na zachytenie axióm viažúcim sa ku konceptom v zdrojovom kóde.

Vedúci: doc. RNDr. Robert Lukoťka, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 31.10.2022

Dátum schválenia: 01.11.2022

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: V prvom rade sa chcem poďakovať môjmu školiteľovi doc. RNDr. Robertovi Lukoťkovi PhD., ktorý mi poskytol rady k práci a pomohol vždy, keď som o to požiadal. Ďalej sa chcem poďakovať mojej rodine za morálnu podporu a poskytnutý priestor pre zvládnutie tejto práce. V neposlednom rade sa chcem poďakovať kamarátom, ktorí ma taktiež podporovali a povzbudzovali k úspešnému dokončeniu tejto práce.

Abstrakt

Generické programovanie je silným nástrojom na písanie udržiavateľného a viacúčelového kódu vďaka parametrizovaniu dátových typov. Z viacúčelovosti kódu nevyplýva, že sa bude správať správne a podľa predpokladov pre všetky typy parametrov – pre správne fungovanie generického kódu je nutné klásť na parametre generického kódu požiadavky. V jazyku C++ je od 90-tych rokov generické programovanie podporované šablónami, ale písanie požiadaviek na parametrizované typy malo až donedávna rôzne nevýhody: nedostatočné vymedzenie, náročnosť písania podmienok alebo náročná diagnostika chybných výpisov kompilátora. Tieto problémy má riešiť jazykový konštrukt koncept, ktorého zavedenie v štandarde C++20 má riešiť spomenuté problémy predchádzajúcich spôsobov na vymedzovanie typov parametrov viacúčelového kódu. Práca poskytuje prehľad o aktuálnom stave konštruktu koncept: motivácia používať koncepty vo viacúčelovom kóde, ale aj potenciálne problémy s jeho využitím; spôsoby zápisu konceptu v zdrojovom kóde; diagnostika kompilačných chýb a rozbor niektorých predefinovaných konceptov z konceptovej knižnice, ktorá je súčasťou Standard Template Library (STL) a porovnanie konceptov s doterajšími spôsobmi na vymedzovanie parametrov generického kódu. V práci taktiež ukážeme, akým spôsobom, respektíve, akých pravidiel sa držať pri písaní a používaní konceptov tak, aby spĺňali zadanie, pre ktoré boli do štandardu zavedené.

Kľúčové slová: C++, generické programovanie, koncepty, vymedzovanie typov

Abstract

Generic programming is a powerful tool for writing maintainable and multi-purpose code thanks to the parametrization of data types. The multi-purpose nature of the code does not imply that it will behave correctly and according to assumptions for all types of parameters – for the proper functioning of the generic code, requirements must be placed on the parameters of the generic code. Since the 1990s, generic programming in C++ has been supported by templates, but until recently, writing requirements for parametrized types had various disadvantages: lack of definition, difficulty in writing conditions, or difficult diagnostics of compiler error listings. These problems are to be solved by the language construct concept, the introduction of which in the C++20 standard is to solve the mentioned problems of the previous methods for defining the types of parameters of multi-purpose code. The work provides an overview of the current state of the concept construct: motivation to use concepts in multi-purpose code, but also potential problems with its use; ways of writing the concept in the source code; diagnosis of compilation errors and analysis of some predefined concepts from the concept library; which is part of the Standard Template Library (STL), and comparison of concepts with existing methods for constraining generic code parameters. In the work, we will also show how, or what rules to follow when writing and using concepts so that they fulfill the task for which they were introduced into the standard.

Keywords: C++, generic programming, concepts, type constraints

Obsah

Zoznam obrázkov

Úvod

Genericita (viacúčelovosť) v programovaní je silným nástrojom pre písanie čistého kódu — vďaka parametrizovaniu typov sme schopní zachytávať myšlienky v kóde všeobecnejšie a dôkladnejšie, eliminujúc potrebu písania sémanticky rovnakých, ale syntakticky rôznych častí kódu, a teda sa riadiť princípom DRY (*Don't Repeat Yourself*). Človek má prístup k písaniu generických algoritmov vo viacerých programovacích jazykoch: za zmienku stoja programovacie jazyky ako Java, Python, Haskell alebo v C++ pod názvom *templates* (šablóny), ktoré budú súčasťou tejto práce. Šablónový kód so sebou nesie jedno riziko – nemôžeme očakávať, že do neho budeme vedieť ľubovoľne vkladať rôzne hodnoty. Vzhľadom na vnútornú implementáciu nemusia byť vkladané argumenty kompatibilné s kódom, čo vyústi do chybových výpisov kompilátora, a teda je vhodné vymedziť prístup k funkcionalite len vyhovujúcej množine argumentov. To vieme dosiahnuť viacerými spôsobmi, od jednoduchších, ktoré dokážu zachytiť aj amatérski programátori, až po komplexné jazykové konštrukty jazyka C++.

Niektoré z nich popíšeme v prvej kapitole a ukážeme, že každá z implementácií nesie svoje nevýhody, ako napríklad náročnosť písania požiadaviek (podmienok) pre dátové typy zo strany programátora, zložitosť chybových výpisov alebo zahltenie obsahu kódu podmienkami, ktoré priamo nesúvisia s funkcionalitou šablónového kódu. Tieto problémy má riešiť `koncept` (menovaná množina požiadaviek), ktorý je oficiálnou súčasťou jazyka C++ od štandardu C++20, a ktorý je témou tejto práce.

V druhej kapitole, okrem stručnej histórie konceptu, vyjadríme, čo je to koncept presne a akú ma syntax. Ukážeme si, akými spôsobmi vieme koncepty implementovať v rámci kódu, kedy je ich využitie vhodné a kedy zase nie. V jednej z podkapitol si prejdeme cez chybový výpis kompilátora za využitia konceptu, ku koncu druhej kapitoly si spomenieme zopár predimplementovaných konceptov z knižnice *concepts* a na záver druhej kapitoly spravíme porovnanie spôsobov na vymedzenie typov.

V tretej kapitole uvedieme usmernenia na používanie a písanie vhodných konceptov tak, aby spĺňali zmysel, pre ktorý boli do štandardu C++20 zavedené.

Cieľom práce je oboznámiť čitateľa s jazykovým konštruktom `koncept` ako jednou z možností na vymedzovanie dátových typov, poskytnúť mu prehľad jeho použitia či už v teoretickej rovine (ukážkou jeho rôznych syntaktických zápisov), ale aj v praktickej rovine (ukážkou viacerých príkladov implementovaných konceptov). Čitateľ by mal po

prečítaní práce rozumieť tomu, ako koncept funguje a jeho význam pre programovanie najmä knižničného kódu.

Kapitola 1

Spôsoby obmedzenia typov v C++

Programovací jazyk C++ je pomerne bohatý z hľadiska možností výberu sémanticky rovnakých funkcionalít, ale s rôznymi syntaxami (prípadne s rôznymi výhodami a nevýhodami pre príslušné spôsoby zápisu). Najprv uvedieme čitateľa do problematiky podkapitolou o generickom programovaní, ktoré je podstatou využívania konceptov a následne ukážeme viacero spôsobov zaručenia vymedzení parametrizovaných typov bez využitia konceptov. Tieto spôsoby sú stále validné, ale majú rôzne nevýhody, a preto boli v štandarde ISO/IEC 14882:2020 (C++20) nahradené konceptami, o ktorých si povieme v kapitole 2, a ktoré sú témou tejto práce.

1.1 Generické programovanie

Generické programovanie je štýl programovania, pri ktorom sú triedy, respektíve funkcie a algoritmy, implementované všeobecne: sú sémanticky správne pre viaceré dátové typy a inštancia pre konkrétne typy parametrov sa vytvorí pre špecifikovaný typ, ktorý je potrebný v rámci behu programu. Tento model je teda založený na abstrakcii – zachytení vzoru alebo podobnosti pre ľubovoľný parameter, v istom zmysle matematicky, a tak, ako si algoritmy predstavujeme v pseudokóde, v ktorom sa primárne sústredíme na samotný algoritmus a zanedbávame detaily [21] [23].

1.1.1 História generického programovania

Generické programovanie bolo prístupné programátorom po prvýkrát v roku 1973 v rámci funkcionálneho jazyka ML a neskôr v roku 1977 sa stalo plnohodnotnou súčasťou jazyka Ada [24]. Do jazyka C++ bola táto možnosť programovania pridaná v roku 1998 v štandarde C++98 (viac v podkapitole 1.2). Samotný pojem „generické programovanie" sa viaže k programátorom Davidovi Musserovi a Alexandrovi Stepanovovi, ktorí spolu v roku 1989 vydali prácu pod názvom *Generic Programming* [25], a v ktorej duchu Stepanov začal okolo roku 1992 navrhovať Štandardnú šablónovú knižnicu

jazyka C++ (*C++ Standard Template Library*, alebo skrátene STL). V súčasnosti je generické programovanie súčasťou viacerých programovacích jazykov na implementáciu viacúčelového kódu.

1.1.2 Výhody a nevýhody abstrakcie

Zoberme si napríklad algoritmus pre usporiadanie množiny prvkov (bez ujmy na všeobecnosti, nech je to usporiadanie od najmenšieho prvku po najväčší). Očakávame, že týmto algoritmom vieme, pre jednoduchosť, usporiadať ľubovoľnú množinu čísel. Z pohľadu programátora bez znalosti generického programovania by to znamenalo, že jeho úlohou je napísať funkciu `sort` pre každý dátový typ reprezentujúci číslo, ktorých je v programovacích jazykoch mnoho. Očividne to nie je optimálny spôsob programovania – ak by sme mali N dátových typov reprezentujúcich číslo, bolo by potrebné napísať N rovnakých kópií funkcie `sort`, pretože všetky typy čísel majú spoločnú vlastnosť – pri ich porovnávaní, na ktorom je založené usporiadanie, sa hľadí na ich hodnotu. Táto duplicita metód by sa ešte viac prejavila pri písaní viacúčelového knižničného kódu.

Predpokladajme, že máme N dátových typov a M funkcií využívajúce dané dátové typy – bez abstrakcie by to vyústilo do $N \cdot M$ počtu funkcií namiesto M funkcií za použitia generického programovania. Hlavnou výhodou generického programovania je teda redukcia duplicitného kódu, s čím je spojená aj čitateľnosť a jednoduchosť orientácie sa v zdrojovom kóde pre programátora.

Predstavme si, že by sme chceli naše, už existujúce, knižničné funkcie aktualizovať. Namiesto úpravy jednej generickej funkcie by programátor musel robiť zmeny v každej z N verzií funkcie, čo pre neho môže byť časovo zaťažujúce – ďalšou výhodou je teda efektívnosť aktualizácie.

V úvode podkapitoly 1.1 sme spomenuli, že písanie viacúčelového kódu je založené na abstrakcii (hľadaní alebo definovaní vzoru spoločného pre všetky dátové typy, pre ktoré má byť funkcia určená). Predpokladajme, že v našej knižnici funkcií s N kópiami funkcie `sort` pre N dátových typov reprezentujúcich číslo sme našli chybu v pôvodnej verzii funkcie `sort`, na ktorej sú založené zvyšné ekvivalentné funkcie. Okrem identifikácie chyby (či už syntaktickej alebo sémantickej) a jej opravy v pôvodnej funkcii je programátor donútený opraviť aj zvyšných N kópií funkcie, čo opäť nie je vyhovujúci spôsob programovania. Pri implementácií funkcie viacúčelovým charakterom programátor identifikuje chybu raz, opraví ju a funkcia bude následne fungovať správne pre všetky dátové typy, čo je ďalšou výhodou tohto spôsobu programovania.

Z viacúčelovosti funkcií (ako napríklad v prípade funkcie `sort`) je zrejmé, že ich použitie nie je vyhradené na jeden konkrétny projekt, ale sú využiteľné univerzálne. Tieto funkcie sú napísané raz a v budúcnosti sú zavádzané do nových programov z predpísaných knižníc, čo šetrí programátorovi čas a necháva ho upriamovať pozornosť

na časti programu, ktoré sú originálne a priamo súvisiace s jeho úlohou.

Nevýhodou generického programovania, konkrétne šablón v jazyku C++, bola napríklad zložitosť písania vymedzení (podmienok, respektíve požiadaviek) na parametrizované typy a diagnostika chybových výpisov kompilátora pri ich porušení – riešením týchto nevýhod majú byť koncepty. Podrobnejšie si o tejto problematike povieme v nasledujúcich podkapitolách.

1.2 Šablóny v C++

Šablóny (`templates`) slúžia ako podpora generického programovania v jazyku C++ od štandardu C++98. V jazyku C++ rozoznávame viaceré typy šablón, najčastejšie sú v praxi používané šablóny tried (`class templates`) alebo šablóny funkcií (`function templates`) [19], na ktoré sa, pre jednoduchosť a stručnosť ukážky, pozrieme v tejto podkapitole. Ako príklad šablónovej funkcie zvolíme funkciu `min` zo Štandardnej šablónovej knižnice jazyka C++ [4], spomenutej v podkapitole 1.1.1, ktorej návratová hodnota je *menší* z argumentov funkcie (pre ktoré je definovaný operátor <) – jej implementácia vyzerá nasledovne:

Algoritmus 1.1: Šablónová funkcia min zo Štandardnej šablónovej knižnice

```
template<class T>
const T& min(const T& a, const T& b)
{
        return a > b ? b : a;
}
```

Viacúčelovosť funkcie je definovaná časťou `template<class T>` – kľúčové slovo `template` symbolizuje, že funkcia `min` je šablónová a `class T` definuje, že aspoň jeden z argumentov funkcie je parametrizovaného typu (triedy) T (v tomto prípade oba). Návratovou hodnotou je hodnota typu T. Kľúčové slovo `class` v hlavičke šablóny môže byť nahradené kľúčovým slovom `typename`, tieto kľúčové slová sú v hlavičke šablóny sémanticky ekvivalentné.

Od štandardu C++20 je možné písať šablónový kód aj pomocou kľúčového slova `auto` – zmažeme hlavičku šablóny funkcie a typ T v hlavičke funkcie nahradíme kľúčovým slovom `auto`. Tento spôsob písania šablónovej funkcie sa nazýva skrátená šablóna funkcie [10]. Skrátená šablónová funkcia `min` zo Štandardnej šablónovej knižnice by vyzerala takto:

Algoritmus 1.2: Skrátená šablónová funkcia min zo Štandardnej šablónovej knižnice

```
const auto& min(const auto& a, const auto& b)
{
        return a > b ? b : a;
}
```

1.3 Dôvody vymedzenia typov

Keď chceme písať šablónový kód, očakávame, že nie všetky parametre, ktoré budeme do daného kódu dosádzať, budú kompatibilné s vnútornou implementáciou častí kódu. Môže nám to buď na výstup vrátiť kompilačnú chybu, ak je daný parameter typu, ktorý nepodporuje operácie v implementácii, alebo ich podporuje, ale nepracuje tak, ako by sme očakávali. V prvom prípade, teda pri použití argumentov nevhodného typu, by sme taktiež chceli dobrú diagnostiku chýb kompilátora — niečo, čo bolo po zavedení šablón v štandarde C++98 postrachom programátorov v C++. Problém, pri ktorom nám kompilátor chybu nevyhodí, ale program nefunguje tak ako by mal, je obtiažnejší na vyriešenie, a môže človeka donútiť stráviť hodiny študovaním internej implementácie knižničných funkcií, čomu by sa dalo predísť sémantickými požiadavkami na typové parametre kódu. Ukážeme si jednoduchú funkciu, ktorej úlohou bude vykonávať súčet dvoch čísel. Keďže typov pre čísla máme niekoľko, nebudeme využívať funkčný *overloading* (definovanie viacerých funkcií s rovnakým menom, ale rôznymi parametrami; preťaženie), ale využijeme šablóny. Funkcia `add_numbers` môže vyzerať napríklad takto:

Algoritmus 1.3: Šablónová funkcia na výpočet súčtu dvoch čísel

```
template<typename U, typename V>
auto add_numbers(U first, V second)
{
        return first + second;
}
```

Poznámka: pre jednoduchosť funkcie `add_numbers` sme ako návratovú hodnotu zvolili `auto`, ktorá je súčasťou až od verzie C++11 [8] – v prípade, že oba typy U a V sú celé čísla, návratovou hodnotou bude taktiež celé číslo, inak keď by bolo aspoň jedno číslo desatinné, návratovou hodnotou bude desatinné číslo. V štandardoch pred C++11 by sme mohli pre jednoduchosť nastaviť návratovú hodnotu na `double`.

Poznámka 2: môžeme povedať, že vhodné ohraničenie funkcie sa dá zaručiť aj vhodným názvom funkcie, ktorý presne definuje význam – k tomuto faktu nie je potrebné

písanie samostatnej podkapitoly a v našej funkcií `add_numbers` sme to už v podstate vyjadrili.

Na túto funkciu budeme postupne aplikovať spôsoby vymedzovania typov U a V tak, aby bolo zaručené, že funkcia naozaj rieši iba súčet dvoch čísel.

1.4 Komentáre

Jedným zo spôsobom, ktorým môžeme aspoň provizórne klásť obmedzenia na typy, je pomocou komentárov. Nie je to ale dostatočne silný nástroj, pretože závisí od toho, či si tieto požiadavky čitateľ prečíta v zdrojovom kóde. Na funkcionalitu programu nemajú žiadny dopad, keďže ich kompilátor ignoruje a nie sú súčasťou *runtime*-u (behu programu).

Algoritmus 1.4: Šablónová funkcia na výpočet súčtu dvoch čísel po pridaní komentáru

```
/* typy U a V by mali byt definovane ako cisla */
template<typename U, typename V>
auto add_numbers(U first, V second)
{
        return first + second;
}
```

Poznámka: na písanie komentárov sú spísané aj odporúčania [27], čo majú komentáre spĺňať; pomocou komentárov vieme taktiež zapisovať komplexnú dokumentáciu, v ktorej môžeme do istej miery zachytiť požiadavky na parametre, pre C++ existuje napríklad nástroj Doxygen [6].

1.5 Operand typeid

Existuje jazykový konštrukt `typeid`, ktorým vieme vykonávať jednoduché porovnávanie typov, a teda aj jednoduché vymedzenie typov počas behu programu [7]. Pre jednoduchosť predpokladajme, že naša šablónová funkcia pracuje s číslami typov `int` alebo `double`. Potom by za použitia operandu `typeid` z knižnice `typeinfo` mohla funkcia `add_numbers` vyzerať takto:

Algoritmus 1.5: Šablónová funkcia na výpočet súčtu dvoch čísel po pridaní testovania typu pomocou operandu typeid

```
#include <string>

template<typename U, typename V>
auto add_numbers(U first, V second)
{
        std::string U_string = typeid(U).name();
        std::string V_string = typeid(V).name();
        if ((U_string.compare(V_string) == 0)
        || ((U_string.compare("i") == 0)
                && (V_string.compare("d") == 0))
        || ((U_string.compare("d") == 0)
                && (V_string.compare("i") == 0)))
        {
                return first + second;
        }
        /* chybny vystup */
        return NaN;
}
```

Očividne tento štýl vymedzenia správania sa funkcie `add_numbers` nie je najvhodnejší: okrem písania podmienok na overenie typov parametrov funkcie, s ktorými pracujeme (a ktorých môže byť viac ako v uvedenom príklade, aj keď by to mohlo porušovať programátorské konvencie na počet argumentov), čo nám zahlcuje telo funkcie, sú tieto podmienky vyhodnocované počas behu programu, a teda nám beh programu môžu zbytočne spomalovať.

1.6 Static assert

`Static assert` (v preklade statické tvrdenie) je jedným zo spôsobov, ako skontrolovať, či sú splnené podmienky (booleovské `constexpr` – výrazy-funkcie, ktoré môžu byť vyhodnotené v kompilačnom čase) v čase kompilácie [11]. Statické tvrdenia sú k dispozícii od štandardu C++11 a v spojení s `type traits` (v preklade typové charakteristiky) môžu byť vhodnou voľbou na vymedzenie parametrizovaných typov, no majú svoje nevýhody. Jednou z nevýhod je nutnosť písať ich do tela funkcie, čo pri veľkom množstve rôznych typov môže zahltiť vnútornú implementáciu, ktorá môže byť sama o sebe veľmi krátka, a od ktorej vďaka statickým tvrdeniam odvrátime pozornosť. Naša funkcia `add_numbers` by s použitím statického tvrdenia (a vhodnej typovej

charakteristiky) mohla vyzerať nasledovne:

Algoritmus 1.6: Šablónová funkcia na výpočet súčtu dvoch čísel po pridaní statického tvrdenia

```
#include<type_traits>

template<typename U, typename V>
auto add_numbers(U first, V second)
{
        static_assert(
        std::is_arithmetic<U>::value
        && std::is_arithmetic<V>::value);
        return first + second;
}
```

V rámci statického tvrdenia sme použili typovú charakteristiku `is_arithmetic`, ktorú pre typ T v kompilačnom čase vyhodnotí na *true*, respektíve *false* práve vtedy, keď je typ T reprezentovaný celočíselne alebo desatinným číslom – vracia teda *true* pre typy ako `int`, `double` alebo `float`, kde je dôvod zrejmý, ale aj napríklad `char`, ktorý je interne reprezentovaný ako celé číslo z intervalu -127 po 127 [17]. Statické tvrdenia a typové charakteristiky sú aj v súčasnosti využívané v internej implementácii konceptov.

1.7 SFINAE

V tejto podkapitole si spomenieme `SFINAE` (*Substitution Failure Is Not An Error*, v preklade „substitučné zlyhanie nie je chyba"), čo je princíp programovania v C++ založený na tom, že ak máme generický kód, ktorého časť s vloženými parametrami nie je skompilovateľná, tak nekončí program automaticky s chybovým výpisom kompilátora na výstupe (tak ako napovedá názov), ale pokračuje sa v hľadaní vhodnej časti generického kódu [9]. Vo svojej podstate nechávame kompilátor rozhodnúť, ktorú časť (funkciu alebo triedu) použiť pre vstupné parametre nejakého dátového typu. Tento princíp sa používal spoločne s jazykovým konštruktom `enable_if` (v preklade „povoľ ak (platí)") v najväčšej miere na vymedzenie typov pre šablónový kód pred zavedením konceptov v štandarde C++20. O detailoch konštruktu `enable_if` sa môžete dočítať v dokumentácií `cppreference` [14]. Ukážeme si jednoduchý príklad programu využívajúceho funkčné preťaženie šablónových funkcií, princípu `SFINAE` a použitia konštruktu `enable_if` pre usmernenie výberu vhodnej funkcie pre kompilátor:

Algoritmus 1.7: Program využívajúci enable_if a princíp SFINAE

```
#include <iostream>
#include <type_traits>

template<typename T>
std::enable_if_t<std::is_arithmetic_v<T>, void> function(T a) {
    std::cout << a * a << std::endl;
}

template<typename T>
std::enable_if_t<!std::is_arithmetic_v<T>, void> function(T a) {
    std::cout << "NaN" << std::endl;
}

int main() {
    function(3); //9
    function(3.5); //12.25
    function('a'); //9409 (ASCII hodnota 'a' je 97, 97*97)
    function(true); //1
    function("abc"); //NaN
}
```

Šablónovú funkciu `function` sme pomocou konštruktu `enable_if` definovali tak, že v prípade, že typ argumentu funkcie T spĺňa typovú charakteristiku `is_arithmetic` (typ T je číslo), tak na výstup vypíše jeho druhú mocninu. Ak argument funkcie nie je typu, ktorý by spĺňal danú typovú charakteristiku, tak na výstup vypíše `NaN` (*Not a Number*, v preklade „nie je to číslo"). Vo funkcii `main` máme napísaných niekoľko vstupov pre preťaženú funkciu `function`, a keďže prvé štyri zo vstupov sú číselného charakteru (postupne: celé číslo; desatinné číslo; znak, ktorý je interne reprezentovaný ako celé číslo; booleovská hodnota, ktorá je interne reprezentovaná číslami 0 alebo 1), kompilátor úspešne nájde vhodnú funkciu a vytvorí jej inštanciu po doplnení argumentov (kompilátor mohol v tomto procese vyskúšať aj druhú verziu funkcie pre nečíselné vstupy, ale z princípu `SFINAE` nebola výsledkom tohto pokusu kompilačná chyba, ale pokračovalo sa v nájdení vhodnej verzie funkcie). Pre piaty vstup, ktorý je typu `string`, kompilátor zvolil druhú funkciu a na výstupe programu skončilo `NaN`.

Nevýhodou tohto spôsobu písania obmedzení na dátové typy je jeho náročnosť, pretože má množstvo spôsobov zápisu a väčšina z nich využíva netriviálnu syntax, ktorá môže demotivovať programátora skúšať písať podmienky pre parametre šablón týmto spôsobom.

Viac sa o detailoch `SFINAE` sa môžete dozvedieť v materiáloch [22] [33].

1.8 Chybový výpis kompilátora

V tejto kapitole si ukážeme, ako vyzerá kompilačná chyba v prípade, ak za parametre funkcie `add_numbers`, definované rôznymi spôsobmi v predchádzajúcich podkapitolách, doplníme niečo, čo nie je číslo. Tento výpis nám bude slúžiť pre porovnanie s chybovým výpisom kompilátora pre funkciu súčtu čísel s použitím konceptu `Number` (koncept čísla), ktorému sa budeme venovať v samostatnej podkapitole 2.5. Pre príklad zlyhania funkcie súčtu dvoch čísel nám za parameter stačí zvoliť dátový typ, pre ktorý nie je definovaná operácia + (plus) – pre jednoduchosť, zvolíme dátový typ `vector` a spôsob ohraničenia parametrizovaného typu zvolíme statické tvrdenie (ostatné spôsoby majú podobné chybové výpisy). Náš program, ktorý na výstupe vráti výpis kompilačnej chyby, môže vyzerať napríklad takto:

Algoritmus 1.8: Program, ktorý končí kompilačnou chybou

```
#include<vector>
#include<type_traits>

template<typename U, typename V>
auto add_numbers(U first, V second)
{
        static_assert(
        std::is_arithmetic<U>::value
        && std::is_arithmetic<V>::value);
        return first + second;
}

int main()
{
    std::vector<int> first{10, 20, 30};
    std::vector<int> second{1, 2, 3};
    auto assert_result = add_numbers(first, second);
}
```

Napriek tomu, že chybový výpis na nasledujúcom obrázku 1.1 v tomto prípade nie je rozsiahly (rozsiahlejší môže byť v závislosti od použitých dátových štruktúr a počtu porušených podmienok na dátové typy), tak pre programátora, a to najmä neskúseného, môže byť problém identifikovať a analyzovať dôvod chyby.

```
main.cpp:7:2: error: static assertion failed due to requirement
'std::is_arithmetic<std::vector<int, std::allocator<int>>>::value'
        static_assert(
        ^
main.cpp:17:26: note: in instantiation of function template specialization
'add_numbers<std::vector<int>, std::vector<int>>' requested here
    auto assert_result = add_numbers(first, second);
                         ^
main.cpp:10:15: error: invalid operands to binary expression
('std::vector<int>' and 'std::vector<int>')
        return first + second;
               ~~~~~ ^ ~~~~~~
/root/emsdk/upstream/emscripten/cache/sysroot/include/c++/v1/__iterator/move_
iterator.h:170:1: note: candidate template ignored: could not match
'move_iterator' against 'vector'
operator+(typename move_iterator<_Iter>::difference_type __n, const
move_iterator<_Iter>& __x)
^
/root/emsdk/upstream/emscripten/cache/sysroot/include/c++/v1/__iterator/rever
se_iterator.h:219:1: note: candidate template ignored: could not match
'reverse_iterator' against 'vector'
operator+(typename reverse_iterator<_Iter>::difference_type __n, const
reverse_iterator<_Iter>& __x)
^
/root/emsdk/upstream/emscripten/cache/sysroot/include/c++/v1/__iterator/wrap_
iter.h:259:21: note: candidate template ignored: could not match
'__wrap_iter' against 'vector'
__wrap_iter<_Iter1> operator+(typename __wrap_iter<_Iter1>::difference_type
__n, __wrap_iter<_Iter1> __x) _NOEXCEPT
                    ^
2 errors generated.
```

Obr. 1.1: Výpis kompilačnej chyby pre funkciu súčtu s vymedzením typu pomocou statického tvrdenia. Na obrázku je znázornených 19 riadkov chybového výpisu vygenerovaného kompilátorom po spustení programu 1.8. *Poznámka*: čísla riadkov, na ktoré odkazuje kompilátor, nie sú v príklade podstatné, sú z kontextu funkcie pochopiteľné

Kapitola 2

Koncept ako C++20 spôsob kontroly typu

V tejto kapitole sa budeme venovať konceptom, ktoré sa v C++20 štandarde stali plnohodnotnou súčasťou jazyka C++. Okrem histórie a samotnej definície konceptu si ukážeme, akými spôsobmi ich vieme implementovať v rámci kódu, prečo by sme ich mali používať a kedy by sme sa mali pozrieť po alternatívnych možnostiach. Ukážeme si, ako diagnostikovať kompilačnú chybu s využitím konceptu, ukážeme si niektoré z preddefinovaných konceptov z knižnice `concepts` a na záver sa pozrieme na porovnanie konceptov s doterajšími spôsobmi na vymedzovanie typov parametrického kódu.

2.1 História konceptov

Myšlienka konceptov sa datuje k štandardu C++11, no pre svoju zložitosť a v snahe zjednodušiť ich použitie koncovým užívateľom (programátorom) bol ich vývoj posunutý na ich uvedenie v neskorších štandardoch. Koncepty vo svojej plnej verzii, ktoré boli pôvodne naplánované sa uviesť v štandarde C++11, mali zahrňovať tzv. konceptové mapy, ktoré by umožňovali mapovanie operácií jedného typu na ekvivalentné operácie druhého typu (teda pri použití typu X povoliť aj typ Y v prípade, že má syntakticky odlišnú, ale sémanticky ekvivalentnú operáciu) alebo axiómy, ktoré užívateľovi dovoľovali sémanticky zachytávať definície bez dôkazov [5]. Tieto doplnky konceptov sa do štandardu C++20 nedostali. V kapitole 1 sme spomenuli dôvody, prečo by sme chceli obmedziť typy pri písaní generického kódu a akým spôsobom to dosiahnuť. Všetky mali ale nejaké nevýhody: nedostatočné uvedenie požiadaviek, zahltenie implementácie alebo zložitá diagnostika kompilačných chýb pri použití nevhodných argumentov. František Silváši sa vo svojej práci z roku 2014 pod názvom *Syntaktické a sémantické požiadavky na parametrizované typy v C++* [26] zaoberal spôsobom, ako syntakticky a sémanticky zachytiť požiadavky na generické typy v C++. Bol schopný teoreticky

zostrojiť nástroj, ktorý by dokázal spĺňať tento účel – a môžeme tvrdiť, že jeho idea konceptu je veľmi podobná konceptu, ktorý môžeme používať v štandarde C++20 a o ktorom si povieme viac v ďalších podkapitolách.

2.2 Koncept v C++20

Koncept je booleovský predikát, ktorého výsledná hodnota sa počíta v čase kompilácie; je to menovaná množina požiadaviek, ktorá sa viaže k šablónovej hlavičke generických častí kódu a slúži ako rozhranie upresňujúce použitie týchto parametrizovaných typov [2]. Tento jazykový konštrukt má plnohodnotne nahradiť metódy z kapitoly 1, rieši ich nevýhody a pár z jeho výhod (o ktorých si povieme viac v nasledujúcich podkapitolách) sú:

- kontrola typu premenných v kompilačnom čase
- užívatelsky prívetivá diagnostika kompilačných chýb
- podpora preťaženia (*overloading*) na základe požiadaviek
- podpora presného zachytávania požiadaviek pre užívateľa (programátora) a vložiť ich v rámci rozhrania (hlavičky)

2.2.1 Syntax konceptu

Syntax konceptu v štandarde C++20 vyzerá nasledovne:

template<*template-parameter-list*>
concept *concept-name* = *constraint-expression*;

V prvom riadku môžeme vidieť kľúčové slovo `template`, ktoré sa používa v hlavičkách generického kódu a `template-parameter-list`, pod ktorým si môžeme predstaviť napríklad triedy alebo typy parametrov. V druhom riadku máme priamo definovanie konceptu cez kľúčové slovo `concept` a jeho meno `concept-name`, následne `constraint-expression`, výraz, ktorý definuje požiadavky pre typy definované v `template-parameter-list`. Keďže koncept je podľa definície booleovský predikát, požiadavky pre typy sú vyhodnocované na *true* alebo *false*, a môžeme očakávať, že na ňom môžeme robiť booleovské operácie konjunkcie, disjunkcie a negácie. Množina požiadaviek v `constraint-expression` môže byť tvorená inými konceptami, typovými charakteristikami, `constexpr` alebo si môžeme definovať vlastné koncepty pomocou kľúčového slova `requires`. V ďalšej podkapitole si uvedieme zopár príkladov jednoduchých konceptov s rôznymi predpismi množiny požiadaviek a v podkapitole 2.3 si ukážeme, akými spôsobmi ich vieme viazať s parametrizovanými typmi v šablónových funkciách [2].

2.2.2 Jednoduché príklady konceptov

V tejto podkapitole si ukážeme príklady konceptov s množinami požiadaviek rôzneho druhu tak, ako sme to spomenuli v predchádzajúcej podkapitole.

Príkladom konceptu vytvoreného pomocou typovej charakteristiky môže byť napríklad koncept `void_type`, ktorý je splnený práve vtedy, keď je typ T typu `void`. Jeho implementácia vyzerá nasledovne:

Algoritmus 2.1: Koncept void_type

```
#include<type_traits>

template<typename T>
concept void_type = is_void<T>::value;
```

Poznámka: Na písanie konceptov pomocou typových charakteristík je potrebné na začiatok kódu vložiť hlavičku `type traits`, ktorá je súčasťou `Metaprogramming` knižnice. Ďalšie príklady typových charakteristík, z ktorých môžete vytvárať koncepty nájdete v dokumentácii `cppreference` [20].

V kapitole 1 sme si ukázali jednoduchú funkciu na súčet dvoch čísel a ako na ňu môžeme aplikovať obmedzenia rôznymi jazykovými konštruktami, ktoré poskytuje jazyk C++. Za pomoci už existujúcich konceptov definovaných v knižnici `concepts` si teraz vytvoríme koncept čísla `Number` jednoducho a elegantne napríklad takto:

Algoritmus 2.2: Koncept Number

```
#include<concepts>

template<typename N>
concept Number = std::integral<N> || std::floating_point<N>;
```

Na definovanie konceptu `Number` sme použili knižničné koncepty `integral` a `floating_point`, vďaka ktorým sme presne definovali, čo si predstavujeme pod číslom: dátový typ N reprezentuje číselný typ práve vtedy, keď reprezentuje celé alebo desatinné číslo – ak nie je splnená ani jedna z podmienok, N číslo nie je [15] [16]. Keďže `integral` a `floating_point` sú koncepty, tak pre ne platí, že sú vyhodnotené (je im priradená pravdivostná hodnota) v čase kompilácie programu.

Poznámka: existuje typová charakteritstika `is_arithmetic`, ktorá vyhodnotí, či je typ T celé číslo alebo desatinné (použili sme ju aj v podkapitole 1.6 alebo v príklade 1.7) – koncept `arithmetic`, ktorý by bol sémanticky ekvivalentný s vyššie uvedeným konceptom `Number` v najnovšom štandarde jazyka C++ pridaný nie je.

Poznámka 2: disjunkcie požiadaviek v konceptoch, v našom prípade konceptu `integral` a konceptu `floating_point`, sú vyhodnocované zľava doprava, pričom ak je ľavý dis-

junkt kompilátorom vyhodnotený ako pravdivý, tak sa pravdivosť pravého disjunktu netestuje (ignoruje sa) a disjunkcia je vyhodnotená ako pravdivá [2].

Predstavíme si jednoduchý koncept reprezentujúci dátové štruktúry, ktoré majú definovanú operáciu `pop` (operácia `pop` z objektu definujúceho množinu prvkov vyhodí jeden prvok, štandardne prvý alebo posledný, v závislosti od objektu a jeho implementácie, a vráti jeho hodnotu). Napísať to môžeme za pomoci kľúčového slova `requires` napríklad takto:

Algoritmus 2.3: Koncept hasPop

```
template<typename Container>
concept hasPop = requires (Container C)
{
        auto h = C.pop();
};
```

Tento koncept reprezentuje všetky typy dátových štruktúr, pre ktoré je definovaná operácia pop: ak je pre parameter *C* definovaná operácia `pop` (teda *C.pop()* je úspešne skompilovateľné), tak je koncept `hasPop` vyhodnotený v čase kompilácie na hodnotu *true*, inak *false*. Napríklad taký `std::stack` alebo `std::vector` tento koncept nespĺňajú, pretože majú iba operáciu `pop_back`, ale `std::list` koncept spĺňa, lebo má definovanú operáciu `pop`.

Poznámka: tu by sa nám hodili konceptové mapy (spomenuté v predchádzajúcej podkapitole) – mohli by sme vytvoriť konceptovú mapu, vďaka ktorej by boli prijateľné aj dátové štruktúry s operáciou `pop_back`.

Na záver si ukážeme, že môžeme písať koncepty vnorenými `requires` výrazmi. Kým použitie jedného `requires` pri definovaní konceptu nám testuje, či sú dané časti kódu skompilovateľné, vnorené `requires` slúži na zachytenie výrazov, ktorých pravdivosť sa má vyhodnotiť v čase kompilácie. Ukážeme si to na jednoduchom príklade konceptu `atleast_int`, ktorý reprezentuje typy celých čísel, ktoré v pamäti zaberajú aspoň 4 byty:

Algoritmus 2.4: Koncept atleast_int

```
#include <concepts>

template<typename T>
concept atleast_int = requires (T a) {
    requires std::integral<T>;
    requires sizeof(a) >= 4;
};
```

Predpísaný význam konceptu sme dosiahli dvomi vnorenými `requires` výrazmi, kde jeden výraz vyjadruje podmienku, či typ T spĺňa koncept `integral` (a teda je celé číslo) a druhý výraz vyjadruje podmienku, či je premenná typu T reprezentovaná v pamäti aspoň 4 bytami. Obe tieto podmienky sú vďaka vnoreným `requires` výrazom vyhodnocované v čase kompilácie.

2.3 Viazanie konceptov na parametrizované typy

V tejto podkapitole sa pozrieme na spôsoby, akými vieme koncepty viazať na parametrizované typy a aké sú rozdiely medzi rôznymi štýlmi viazania konceptov. Materiály, z ktorých sme informácie čerpali, a na ktorých sa môžete dozvedieť viac, sú uvedené v závere podkapitoly.

Za koncept, ktorým budeme viazať parametrizované typy v nasledujúcich príkladoch, zvolíme koncept čísla `Number`, ktorý sme definovali v predošlej podkapitole. Pre jednoduchosť príkladu budeme pracovať s generickou funkciou, na ktorú aplikujeme koncept `Number` – funkciou `add_numbers`, ktorá zo vstupných argumentov ľubovoľného typu vráti ich súčet, a ktorú sme používali na ukážku starších spôsobov viazania typov v kapitole 1.

Jedným zo spôsobov ako aplikovať koncept `Number` na funkciu `add_numbers` je nahradiť kľúčové slovo `typename` názvom konceptu v hlavičke šablóny funkcie. Implementované v kóde to bude vyzerať nasledovne:

Algoritmus 2.5: Šablónová funkcia s aplikovaním konceptu nahradením kľúčového slova typename

```
#include<concepts>

template<typename N>
concept Number = std::integral<N> || std::floating_point<N>;

template<Number U, Number V>
auto add_numbers(U a, V b)
{
    return a + b;
}
```

Ďalším zo spôsobov aplikácie konceptu na šablónovú funkciu je pomocou kľúčového slova `requires` tak, že ho doplníme spoločne s aplikovaným konceptom za hlavičku šablóny funkcie takto:

Algoritmus 2.6: Šablónová funkcia s aplikovaním konceptu využitím requires

```
#include<concepts>

template<typename N>
concept Number = std::integral<N> || std::floating_point<N>;

template<typename U, typename V>
requires Number<U> && Number<V>
auto add_numbers(U a, V b)
{
    return a + b;
}
```

Všimneme si, že vzhľadom na to, že argumenty funkcie `add_numbers` sú dvoch typov a na oba aplikujeme koncept `Number`, podmienku na typy musíme zapísať v tvare konjunkcie.

Od štandardu C++20 môžeme v hlavičke funkcie nahrádzať typy argumentov kľúčovým slovom `auto`, čím sa z nich stávajú parametre neohraničeného typu, teda sú generické. Tento spôsob písania funkcií sa nazýva skrátená šablóna a spomínali sme si ju na príklade šablónovej funkcie `min` v algoritme 1.2. Aj na túto verziu šablónovej funkcie môžeme aplikovať koncepty, a to vložením názvu konceptu pred kľúčové slovo `auto`, ktoré je pred názvom argumentu funkcie. Najprv si ukážeme skrátenú šablónovú funkciu `add_numbers` a následne jej verziu s aplikáciou konceptu `Number`.

Algoritmus 2.7: Skrátená šablónová funkcia add_numbers

```
auto add_numbers(auto a, auto b)
{
    return a + b;
}
```

Skrátena šablónová funkcia `add_numbers` s aplikáciou konceptu `Number` bude implementovaná nasledovne:

Algoritmus 2.8: Skrátená šablónová funkcia add_numbers s aplikovaním konceptu Number

```
auto add_numbers(Number auto a, Number auto b)
{
    return a + b;
}
```

Koncepty v tomto prípade ale nie sú limitované na argumenty funkcie – v našom programe 2.8 je návratová hodnota „typu" `auto` a z predpisu funkcie (súčet čísel) je zrejmé, že aj návratová hodnota spĺňa koncept `Number`. Štandard C++20 nám dovoľuje aplikovať koncepty aj na neohraničené premenné (`auto`) [8]. Aplikáciou konceptu `Number` aj na návratovú hodnotu dostávame nasledovnú skrátenú šablónovú funkciu `add_numbers`:

Algoritmus 2.9: Skrátená šablónová funkcia add_numbers s aplikovaním konceptu Number aj na návratovú hodnotu

```
Number auto add_numbers(Number auto a, Number auto b)
{
    return a + b;
}
```

Z vyššie spomenutej funkcie si môžeme všimnúť, že použitie konceptu spoločne s kľúčovým slovom `auto` pripomína definovanie typu premennej (v tomto prípade definuje celú množinu typov, ktoré spĺňajú koncept `Number`). V skutočnosti to ale nie je typ premennej, pretože po aplikovaní konkrétnych argumentov do hlavičky funkcie sa určí aj konkrétny typ návratovej hodnoty. Tento spôsob je ale užívateľsky prívetivý z hľadiska dokumentácie správania funkcie, pretože podstata funkcie je jednoducho čitateľná.

Viac sa o možnostiach využitia konceptov na parametrizované typy môžete dozvedieť v materiáloch [2] [31] [32].

2.4 Motivácia k použitiu konceptov

Motiváciou k použitiu konceptov v praxi sú ich výhody, pričom niektoré z nich sme už spomenuli v podkapitole 2.2. Jednej z nich, diagnostike chybových výpisov kompilátora, je venovaná samostatná podkapitola 2.5.

Pomocou konceptov sme schopní písať čistejší kód, ktorý je významovo precízny a umožňuje sa čitateľovi lepšie zorientovať v zdrojovom kóde. Zoberme si napríklad koncept `Number` z podkapitoly 2.2.2. V kapitole 1 sme si ukázali viacero spôsobov, ako napísať generickú funkciu na súčet dvoch čísel s ohraničením dátových typov – všetky z nich mali robustný zápis funkcie (s výnimkou komentárov, tie ale vymedzenie typov nezaručili). Pomocou konceptu `Number` sme v predošlej kapitole 2.3 boli schopní napísať funkciu `add_numbers` úhľadne a stručne.

Ďalším príkladom môže byť hlavička funkcie `sort` – `void sort(Sortable s)`. Je z nej zrejmé, že funkcia `sort` funguje správne pre parametre spĺňajúce koncept `Sortable`, teda také množiny prvkov, ktoré sa dajú usporiadať. Bjarne Stroustrup vo svojej práci

Concepts: The Future of Generic Programming or How to design good concepts and use them well (v preklade Budúcnosť generického programovania alebo ako navrhovať dobré koncepty a používať ich správne) [28] spomína funkciu `sort` ako dobré použitie konceptu, pretože presne vyjadruje čo je významom funkcie `sort`. Koncept `Sortable` je v knižnici `iterator` definovaný takto [30]:

Algoritmus 2.10: Koncept Sortable

```
template<class Container, class Comparator = ranges::less,
         class Proj = std::identity>
concept sortable = std::permutable<Container> &&
    std::indirect_strick_weak_order<Comp,
         std::projected<Container, Proj>>;
```

Koncepty, ako menovaná množina požiadaviek, môžu slúžiť ako forma dokumentácie pre generické časti kódu. Ak na nejaké generické triedy alebo funkcie viažeme požiadavky vo forme konceptov, programátori majú požiadavky a predpokladané správanie funkcií napísané priamo v zdrojovom kóde, čo môže zefektívniť ich prácu.

Okrem toho, že pomocou konceptov vieme písať precízne podmienky, ktoré čitateľovi uľahčujú porozumenie zdrojového kódu, tak vďaka presnému všeobecnému charakteru zachytenia požiadaviek sú koncepty znovupoužiteľné v ďalších projektoch.

V kapitole 2.2.1 sme uviedli, že vďaka definícií konceptu, ako menovanej množine požiadaviek vyhodnocovanej na booleovskú hodnotu v kompilačnom čase, môžeme v rámci jeho predpisu robiť booleovské operácie. Pomocou nich vieme množinu vytvorených konceptov jednoducho zväčšovať, a to aplikáciou booleovských funkcií na prvky tejto množiny. Konjunkciou konceptov vieme vytvárať špecializovanejšie koncepty, kým disjunkciou konceptov môžeme vytvárať koncepty s rozšírenou splniteľnou množinou typov. Množinu vytvorených konceptov týmto spôsobom môžeme rozširovať bez toho, aby sme menili už existujúci kód.

Zoberme si opäť funkciu `add_numbers`. Vzhľadom na to, že koncept je definovaný ako booleovský predikát, ktorý je vyhodnocovaný v čase kompilácie, kontrola, či parametre spĺňajú koncept `Number` sa odohráva v čase kompilácie a teda nezaťažuje beh programu, čo vyústi v jeho rýchlejšie ukončenie. Síce kompilátoru, v istom zmysle, dáva viac práce, ale v konečnom dôsledku sa to vyváži, a dokonca je to aj efektívnejší spôsob vymedzenia typov ako princíp `SFINAE`, ktorý hľadá vhodného kandidáta pre substitúciu typu za parametrizovaný typ.

2.5 Diagnostika chýb pomocou konceptov

V tejto podkapitole si stručne ukážeme, akým spôsobom vieme pomocou konceptov jednoducho diagnostikovať chyby, ktoré na výstup vracia kompilátor. Využijeme pred-

pis funkcie `add_numbers` z programu 2.5 a spojíme ho s funkciou `main` programu 1.8 z predchádzajúcej kapitoly, na ktorom sme testovali chybový výpis kompilátora s vymedzením parametrizovaného typu pomocou statického tvrdenia. Náš program bude vyzerať takto:

Algoritmus 2.11: Program s konceptom, končiaci kompilačnou chybou

```
#include<concepts>
#include<vector>

template<typename N>
concept Number = std::integral<N> || std::floating_point<N>;

template<Number U, Number V>
auto add_numbers(U a, V b)
{
    return a + b;
}

int main()
{
 std::vector<int> first{10, 20, 30};
 std::vector<int> second{1, 2, 3};
 auto result = add_numbers(first, second);
}
```

Na nasledovnom obrázku 2.1 vidíme, že napriek tomu, že v tomto konkrétnom príklade je chybový výpis kompilátora pri použití konceptu `Number` dlhší ako pri použití statického tvrdenia na funkciu `add_numbers` v programe 1.8, tak je jednoducho čitateľný – chybový výpis znie: neexistuje vhodná funkcia pre volanie `add_numbers`, pretože požiadavky pre typy nie sú splnené (pre typ U `vector` a typ V `vector`), pretože `std::vector<int>` nespĺňa koncept `Number`, pretože nespĺňa koncept `integral` (koncept, ktorým definujeme koncept `Number`), a to pretože typová charakteristika `is_integral_v` je vyhodnotená na *false* (tu končí strom vyhodnocovaní). Rovnako `std::vector<int>` nespĺňa koncept `floating_point`, pretože typová charakteristika `is_floating_point_v` bola vyhodnotená na false. Teda nebola splnená disjunkcia v definícii konceptu `Number`, pretože koncepty `integral` a `floating_point` neboli pre typ `vector` splnené.

```
main.cpp:17:16: error: no matching function for call to 'add_numbers'
 auto result = add_numbers(first, second);
               ^~~~~~~~~~~
main.cpp:8:6: note: candidate template ignored: constraints not satisfied
[with U = std::vector<int>, V = std::vector<int>]
auto add_numbers(U a, V b)
     ^
main.cpp:7:10: note: because 'std::vector<int>' does not satisfy 'Number'
template<Number U, Number V>
         ^
main.cpp:5:18: note: because 'std::vector<int>' does not satisfy 'integral'
concept Number = std::integral<N> || std::floating_point<N>;
                 ^
/root/emsdk/upstream/emscripten/cache/sysroot/include/c++/v1/__concepts/arith
metic.h:26:20: note: because 'is_integral_v<std::vector<int> >' evaluated to
false
concept integral = is_integral_v<_Tp>;
                   ^
main.cpp:5:38: note: and 'std::vector<int>' does not satisfy 'floating_point'
concept Number = std::integral<N> || std::floating_point<N>;
                                     ^
/root/emsdk/upstream/emscripten/cache/sysroot/include/c++/v1/__concepts/arith
metic.h:35:26: note: because 'is_floating_point_v<std::vector<int> >'
evaluated to false
concept floating_point = is_floating_point_v<_Tp>;
                         ^
1 error generated.
```

Obr. 2.1: Výpis kompilačnej chyby pre funkciu súčtu s ohraničením pomocou konceptu Number. Na obrázku je znázornených 22 riadkov chybového výpisu vygenerovaného kompilátorom po spustení programu 2.11. *Poznámka*: čísla riadkov, na ktoré odkazuje kompilátor, nie sú v príklade podstatné, sú z kontextu funkcie pochopiteľné

2.6 Nevýhody konceptov

V tejto podkapitole si spomenieme niektoré vlastnosti konceptov, ktoré môžu pre programátorov, ktorých cieľom je implementovať koncepty vo svojej práci, spôsobovať ťažkosti, a ktoré by si mal programátor pred použitím konceptov uvedomiť.

Jednou z nevýhod konceptov je, že na písanie vlastných zložitých a komplexných požiadaviek pre dátové typy vo forme konceptov je programátor donútený poznať náročnú funkcionalitu jazyka C++, a to, okrem iných, typové charakteristiky, pomocou ktorých sú koncepty interne implementované. V tomto prípade koncepty používateľovi neposkytujú to, od čoho boli vytvorené, a teda jednoduché zachytenie požiadaviek pre dátové typy – programátor bude schopný vytvárať len koncepty prostredníctvom iných, napríklad tých, ktoré sú definované v knižnici *concepts*, ktorej sa venuje podkapitola 2.7. Náročnostný skok sme si mohli všimnúť aj pri využití vnoreného výrazu `requires requires` pri definovaní konceptu `atleast_int` v kapitole 2.3.

Integrovanie konceptov do kódu využívajúce tradičné postupy na vymedzenie typov nemusí byť jednoduché. V prípade, že niektoré časti kódu nepodporujú využitie konceptov, tak ich použitie vo väčšej miere nie je výhodné – kombinovanie konceptov a iných spôsobov na vymedzenie dátových typov môže negatívne pôsobiť na udržiavateľnosť kódu.

Vzhľadom na to, že koncepty sú relatívnou novinkou v jazyku C++ (aj napriek tomu, že je myšlienka konceptov je stará), ich podpora, či už knižničná alebo aj kompilátorová, môže byť limitovaná, čo môže vyústiť do nepríjemností.

2.7 Knižnica concepts

V tejto podkapitole spomenieme niektoré z preddefinovaných konceptov z knižnice `concepts`. Táto podkapitola je založená na dokumentácií `cppreference` [3], kde si čitateľ môže o knižničných konceptoch prečítať viac. Knižnica `concepts` ponúka preddefinované koncepty viažuce sa či už na základné jazykové požiadavky, objektové požiadavky alebo požiadavky pre porovnávanie, ktoré môžu byť vyhodnocované v čase kompilácie. Na to, aby programátor mohol využívať koncepty z knižnice `concepts`, musí ju do svojho kódu zahrnúť v hlavičke príkazom `#include <concepts>`. Niektoré z knižničných konceptov sme použili v predchádzajúcich podkapitolách – konkrétne koncepty `integral` a `floating_point` pri definovaní konceptu čísla `Number` v kapitole 2.2.2. V dokumentácií sú implementované pomocou typových charakteristík takto:

Algoritmus 2.12: Koncepty integral a floating_point

```
#include <type_traits>

template<typename T>
concept integral = std::is_integral<T>::value;

template<typename T>
concept floating_point = std::is_floating_point<T>::value;
```

Jeden z konceptov definovaných v knižnici `concepts` je koncept `same_as`, ktorý je splnený pre 2 dátové typy U a V práve vtedy, keď je dátový typ U rovný dátovému typu V a naopak [18]. Jeho implementácia môže byť napríklad takáto:

Algoritmus 2.13: Koncept same_as

```
#include <type_traits>
template<typename U, typename V>
concept same = std::is_same_v<U, V>;

template<typename U, typename V>
concept same_as = same<U, V> && same<V, U>;
```

Príkladom použitia konceptu `same_as` môže byť, ak chceme v rámci generického kódu odlíšiť správanie algoritmu pre rovnaké parametrizované dátové typy od všeobecného algoritmu, teda mať v tomto prípade dve funkcie:

Algoritmus 2.14: Príklad použitia konceptu same_as

```
#include <concepts>

template<typename U, typename V>
auto generalized_function(U a, V b)
{
    /* algoritmus generalizovanej funkcie */
}

template<typename U, typename V>
requires std::same_as<U, V>
auto specialized_function(U a, V b)
{
    /* algoritmus specializovanej funkcie */
}
```

Poznámka: V tomto príklade sme pre ilustračné účely zvolili názvy funkcií tak, aby jasne reprezentovali, ktorá funkcia je pre parametre ľubovoľných dátových typov a ktorá je špecializovaná. V praxi by sme mohli nechať názov funkcie rovnaký (napríklad `function`), využiť funkčné preťaženie a výber vhodnej funkcie pre vstupné parametre nechať na kompilátor.

Ďalším z konceptov je napríklad koncept `derived_from`. Je to koncept, ktorý je splnený pre dva dátové typy `Derived` a `Base` práve vtedy, keď je typ `Derived` odvodený od typu `Base` – inými slovami, ak je typ `Base` základom pre typ `Derived` a typ `Derived` je konvertovateľný na `Base` [13]. V rámci knižnice `concepts` je implementovaný nasledovným spôsobom:

Algoritmus 2.15: Koncept derived_from

```
template<class Derived, class Base>
concept derived_from = std::is_base_of_v<Base, Derived>
        && std::is_convertible_v<const volatile Deprived*,
                const volatile Base*>;
```

Vďaka tomuto konceptu teda vieme písať parametrizovaný kód, ktorý bude kompilovateľný len pre tie dátové typy, ktoré sú podtypom iného typu, na ktorý sa daný koncept vzťahuje. Jednoduchým príkladom môže byť trieda `Zviera` a jej podtriedy `Slon` a `Žirafa`. Každá z týchto tried má definovanú funkciu `zakrič` s vlastným originálnym výstupom – ak napíšeme generickú funkciu `zvierací_krik`, ktorej argument `arg` je parametrizovaného typu T, ktorý je vymedzený konceptom `derived_from`, pričom `Base` triedou je trieda `Zviera` a v rámci funkcie `zvierací_krik` voláme funkciu `zakrič` pre argument `arg`, potom inštancie triedy `Slon` a `Žirafa` budú pre funkciu `zvierací_krik` vracať príslušný výstup vzhľadom na svoje implementácie funkcie `zakrič`, ale pre dátové typy (triedy), ktoré nie sú podtriedami triedy `Zviera`, kompilátor vráti chybové hlásenie.

2.8 Porovnanie spôsobov vymedzovania typu

V tejto podkapitole porovnáme koncept a jeho vlastnosti, ktoré sme popísali v predchádzajúcich podkapitolách kapitoly 2 s vlastnosťami jazykových konštruktov popísaných v kapitole 1. Porovnanie s komentármi v tejto podkapitole nebude spomenuté, pretože komentáre nezaručujú skutočné vymedzenie dátových typov parametrizovaného kódu.

Definícia konceptu z podkapitoly 2.2 znie, že koncept je booleovský predikát, ktorého výsledná hodnota sa počíta v čase kompilácie. To znamená, že kontrola neprebieha počas behu programu, tak, ako sa kontrolujú typy pomocou operandu `typeid`. Vyhodnotenie podmienky (predikátu) kompilátorom je rýchlejšie ako kontrola podmienok po-

čas behu programu. Pre zvyšné spôsoby – statické tvrdenia a `enable_if` v kombinácií so `SFINAE`, sú podmienky kontrolované v čase kompilácie rovnako ako pre koncepty.

V podkapitole 2.5 sme si ukázali, akým spôsobom sa diagnostikuje výpis kompilačnej chyby za pomoci konceptu a zistili sme, že jeho formát môže byť čitateľný aj pre niekoho, kto nie je znalý problematiky. V podkapitole 1.8 sme ukázali funkciu využívajúcu statické tvrdenia a program, ktorý vedie ku kompilačnej chybe. Z obrázka 1.1 popisujúceho výpis kompilačnej chyby daného programu sme usúdili, že napriek kratšej dĺžke chybového hlásenia oproti verzii s konceptom je chybový výpis kompilátora s využitím statického tvrdenia náročnejší na diagnostiku. V práci konkrétne príklady chybových hlásení pre operand `typeid` a konštrukt `enable_if` v kombinácii so `SFINAE` neboli uvedené, dostupné zdroje [1] tvrdia, že na tom nie sú lepšie ako koncepty.

Čo sa týka podpory preťaženia na základe požiadaviek, oba konštrukty koncept aj `enable_if` v kombinácií so `SFINAE` môžu využívať túto funkcionalitu, kým statické tvrdenia sa používajú iba na kontrolu podmienky v kompilačnom čase v rámci funkcie.

Posledným bodom spomenutým v podkapitole 2.2 je podpora presného zachytávania požiadaviek pre užívateľa a vložiť ich v rámci hlavičky (funkcie). Vzhľadom na to, že statické tvrdenia aj operand `typeid` sa nachádzajú vo vnútri funkcie, pri tomto bode sa môžeme zamýšľať opäť len nad konceptami a konštruktom `enable_if` a `SFINAE`. Oboma spôsobmi vieme zachytávať požiadavky na parametrizované typy v hlavičke funkcie, ale v prípade `enable_if` to môže byť svojou syntaxou, ktorú sme ukázali v programe 1.7, náročnejšie na pochopenie pre koncového užívateľa, kým koncepty majú syntax jednoduchšiu.

Okrem týchto vlastností sme v rámci podkapitol kapitoly 1 uviedli nevýhody statických tvrdení a operandu `typeid` – tým bola nutnosť písať podmienky do tela funkcie, čím zahlcovali implementáciu algoritmu funkcie.

Pre konštrukt `enable_if` sme v podkapitole o `SFINAE` 1.7 spomenuli, že nevýhodou tohto typu vymedzovania parametrizovaných typov je náročnosť písania podmienok. V podkapitole 2.3 o viazaní konceptov na parametrizované typy sme ukázali niekoľko príkladov rôznych syntaxí konceptu a v podkapitole 2.7 o knižnici `concepts` sme ukázali príklady knižničných konceptov, o ktorých si dovolíme tvrdiť, že majú jednoduchšiu syntax ako `enable_if`.

Kapitola 3

Smernice na prácu s konceptami

V tejto kapitole uvedieme viacero smerníc, ktorými by sa mali riadiť pri písaní a používaní či už vlastných alebo knižničných konceptov. Smernice sú prevzaté z *C++ Core Guidelines* [29], kde sa čitateľ môže dozvedieť viac detailov.

3.1 Smernice na používanie konceptov

Jednou zo smerníc je, že by sme mali používať koncepty vo väčšine prípadov, keď máme v pláne písať generický kód. Koncept slúži na zvýšenie čitateľnosti zdrojového kódu a overenie jeho správnosti, čo je dôležitým prvkom pri programovaní kvalitného a plnohodnotného kódu. Všimnúť si to môžeme na definíciách funkcií `add_numbers` z príkladov 2.7 a 2.9, i keď v danom príklade sú podstata a význam funkcie zrejmé z jej názvu. Ak zmeníme jej názov na `add`, stane sa z nej nejednoznačná funkcia:

Algoritmus 3.1: Šablónová funkcia add

```
auto add(auto a, auto b) {
    /* algoritmus funkcie add */
}
```

Bez dokumentácie alebo iného popisu fungovania funkcie nevieme určiť, pre aké typy argumentov je táto funkcia určená (bude fungovať podľa predpokladov a cieľa programátora, ktorý danú funkciu písal). Bez znalosti jej vnútornej implementácie môžeme z názvu funkcie uvažovať, aké je jej správanie, pričom máme na výber z aspoň troch možností – je to funkcia `add` pre sčitovanie čísel, funkcia `add` pre zreťazovanie reťazcov alebo funkcia `add` pre pridávanie prvkov do nejakej množiny. Pridaním konceptu pred kľúčové slovo `auto` budeme schopní presne určiť, čo je úlohou danej funkcie.

Ak to nie je nevyhnutné, mali by sme sa vyvarovať používaniu vlastných konceptov, hlavne vtedy, ak je daný koncept už napísaný a uverejnený v niektorej z dôveryhodných knižníc (napríklad koncepty z knižnice `concepts`). Môže to nie len zvýšiť efektívnosť

našej práce eliminovaním písania duplicitných konceptov, ale aj redukovať riziko chyby, pretože overené knižničné koncepty prešli kontrolou správnosti, kým tie naše vlastné nemusia byť správne.

Tak ako sme uviedli v podkapitole 2.3, v prípade návratovej hodnoty funkcie alebo pri deklarácií premennej, kde využívame kľúčové slovo `auto`, je pred neho vhodné vložiť koncept, ktorý môže eliminovať prípadnú chybu, ale hlavne svojím menom dodáva premennej význam. Príklad použitia konceptu `Number`, definovaného v programe 2.2, na špecifikáciu premennej s identifikátorom `auto`:

Algoritmus 3.2: Príklad preferovaného použitia konceptu pri deklarovaní premennej

```
/* cast kodu */
vector vec{1, 2, 3};
auto element_bad = vec.at(1);
Number auto element_good = vec.at(1);
```

V podkapitole 2.3 sme ukázali viacero spôsobov ako využiť koncepty na vymedzenie typov šablónových funkcií. Najoptimálnejšia verzia z hľadiska praktickosti je tá najkratšia, teda uvedená v programe 2.9, pretože je najpodobnejšia spôsobu, akým čítame a vysvetľujeme zdrojový kód.

3.2 Smernice na písanie konceptov

Písanie silných konceptov, ktoré presne definujú množinu typov objektov prostredníctvom množiny požiadaviek nie je jednoduché. Definovanie konceptu pripomínajúceho matematickú definíciu (napríklad v nasledujúcom príklade 3.3) nám ukazuje, že koncepty neboli vytvorené pre účely vymedzenia parametrizovaných typov jedných konkrétnych tried a funkcií – v skutočnosti majú väčší potenciál.

Keďže koncepty majú sémantický význam, je vhodné, aby sme v akejkoľvek forme túto sémantiku pri písaní konceptu zaznamenali, napríklad vo forme komentárov. Ako príklad zvolili autori smerníc koncept definujúceho čísla pre typ T v matematickom zmysle: pre typ T sú definované operácie + (plus), − (mínus), · (krát) a ÷ (deleno), operácia na argumentoch a a b je definovaná (skompilovateľná) a výsledok aritmetickej operácie je konvertovateľný na typ T (použitie konceptu `convertible_to` [12]).

Poznámka: V danom príklade sa môžeme zamyslieť, prečo nie je využitý namiesto konceptu `convertible_to` koncept `same_as`. Môžu existovať typy čísel (špecializované typy), ktorých výsledky po vykonaní jednej z operácií koncept `same_as` nespĺňali, ale boli by konvertovateľné naspäť, teda by spĺňali koncept `convertible_to`. Koncept `same_as` je teda moc reštriktívny a z praktického hľadiska je vhodnejšie využiť koncept `convertible_to`. Časť kódu z textu smerníc:

Algoritmus 3.3: Príklad konceptu Number so sémantickým popisom

```
/* cast kodu */
template<typename T>
// predpokladame, ze operatory +, -, * a / splnaju mat.
// definicie ako komutativnost, distributivnost, ...
concept Number = requires (T a, T b)
{
    {a + b} -> convertible_to<T>;
    {a - b} -> convertible_to<T>;
    {a * b} -> convertible_to<T>;
    {a / b} -> convertible_to<T>;
};
```

Pri písaní konceptov je nutné si uvedomiť, či sémanticky vyjadrujú to, čo od nich požadujeme. Mali by sme sa vyhýbať písaniu konceptov, ktoré nemajú veľký význam a sú v princípe jednoduché, pretože nemusia byť dostatočne precízne a pri ich použití môže dôjsť k nečakaným chybám. Autori smerníc uvádzajú príklad konceptu `Addable`, o ktorého význame sa môžeme zamýšľať, že je určený pre typy čísel, ale pre niektoré typy, ktoré nie sú čísla, tento koncept môže byť splnený a funkcia, na ktorú je koncept aplikovaný, bude mať neštandardné správanie:

Algoritmus 3.4: Príklad konceptu bez presného významu

```
/* cast kodu */
template<typename T>
concept Addable = requires (T a, T b)
{
    a + b;
};

Addable auto sum(Addable auto a, Addable auto b)
{
    return a + b;
}

...
//ocakavane spravanie
std::cout << sum(4, 5) << std::endl;
//neocakavane spravanie
std::cout << sum("a", "b") << std::endl;
```

V tomto príklade je koncept `Addable` splnený aj pre typ `string`, keďže typ `string` má definovaný operátor + (plus) ako zreťazenie, a teda kým by sme očakávali na výpise kompilačnú chybu, v skutočnosti bol vypísaný reťazec *ab*.

V prípade, že chceme napísať šablónovú funkciu s aplikovaným konceptom `Concept` na parametrizovaný typ T a druhú šablónovú funkciu pre všetky tie typy, ktoré koncept `Concept` nespĺňajú, nebudeme to robiť postupom, akým sme to robili pri jazykovom konštrukte `enable_if` z kapitoly 1.7, teda písaním jednej šablónovej funkcie s aplikáciou konceptu `Concept` na parameter T a druhej šablónovej funkcie s aplikáciou konceptu `not Concept` na parameter T. Ak osamostatníme druhú funkciu, jej predpis značí, že príjima argumenty typov, ktoré nie sú typu `Concept` – význam takejto funkcie môže byť nejasný. Riešenie problému je v skutočnosti jednoduchšie: pre jednu funkciu bude na parametrizovaný typ aplikovaný koncept `Concept` a druhá funkcia ostane bez vymedzenia generického typu – v tomto prípade nechávame na kompilátor, ktorú z funkcií pre konkrétny dátový typ argumentu zvolí: kompilátor zvolí funkciu bez vymedzenia len v prípade, že koncept `Concept` nebol pre daný typ v čase kompilácie vyhodnotený na *true*. V kóde by to vyzeralo nasledovne:

Algoritmus 3.5: Príklad komplementárnych funkcií vzhľadom na parametrizované typy

```
template<typename T>
auto fun(T x)
{
    /* algoritmus pre typy nesplnajuce koncept */
}

template<Concept T>
auto fun(T x)
{
    /* algoritmus pre typy splnajuce koncept */
}
```

Poznámka: Postup, ktorý sme aplikovali na jazykovom konštrukte `enable_if`, by sa pri použití viacerých konceptov na vymedzovanie typov značne skomplikoval, pretože už pri dvoch konceptoch by bolo potrebné napísať 4 funkcie s rôznymi vymedzeniami typov a pre viac konceptov by počet funkcií narastal exponenciálne.

Z podkapitoly 2.2.1 o syntaxi konceptu vieme, že koncepty vieme definovať aj pomocou iných konceptov, pretože návratová hodnota konceptu je booleovského typu. Napísanie konceptu pre akúkoľvek triviálnu operáciu a následné využitie tejto množiny konceptov nemusí byť vždy ideálne – aj keď sa to zdá byť na pohľad úhľadnejšie, tak precízny zápis je pre čitateľa pochopiteľnejší. Autori smerníc ako príklad uviedli koncept `Equality`, ktorý je splnený pre tie typy, pre ktoré je definovaná operácia ==

(rovná sa) a operácia ! = (nerovná sa). Tento koncept môžeme zapísať, pre ukážku príkladu, dvomi spôsobmi – ako konjunkciu dvoch konceptov `has_equal` a `has_noequal` alebo predpisom konceptu pomocou kľúčového slova `requires`. Oba spôsoby uvedieme v následujúcej časti kódu:

Algoritmus 3.6: Príklad horšieho a lepšieho definovania konceptu Equality z hľadiska pochopiteľnosti

```
#include <concepts>

/* horsi zapis */
template<typename T>
concept Equality = has_equal<T> && has_noequal<T>;

/* lepsi zapis */
template<typename T>
concept Equality = requires (T a, T b) {
    {a == b} -> convertible_to<bool>;
    {a != b} -> convertible_to<bool>;
};
```

Z hornej deklarácie konceptu `Equality` nemusí byť ihneď zrejmé, ako sú koncepty `has_equal` a `has_noequal` definované interne, čo nám limituje pochopenie samotného konceptu `Equality`, kým spodná deklarácia konceptu `Equality` to poskytuje. V prípade, že sú pre dva argumenty a a b typu T skompilovateľné výrazy `a == b` a `a != b` a v oboch prípadoch je výsledkom týchto výrazov booleaovská hodnota, tak typ T spĺňa koncept `Equality`.

Samozrejme by na názvy konceptov mali byť uplaťnované menovacie konvencie tak, že sa z mena konceptu dá interpretovať množina typov, pre ktoré je koncept splniteľný. Koncept, ktorého názov neodzrkadľuje jeho význam nedáva z praktického hľadiska zmysel.

Záver

V tejto práci sme predstavili koncepty ako plnohodnotnú náhradu statických tvrdení a kombinácie jazykového konštruktu `enable_if` spolu s princípom `SFINAE` na vymedzovanie parametrizovaných typov šablónového kódu v jazyku C++.

Na začiatku práce sme okrem popisu generického programovania, jeho výhod oproti špecializovanému programovaniu pre konkrétne dátové typy a šablón ako realizácie generického programovania v jazyku C++ uviedli niekoľko spôsobov, ako sa vymedzovali parametrizované typy pred uvedením konceptov v štandarde C++20. Na príkladoch funkcií sme ukázali jednotlivé nevýhody týchto spôsobov a v závere poskytli príklad výpisu kompilačnej chyby, ktorej potreba uľahčenia čitateľnosti bola jedným z hlavných cieľov pridania jazykového konštruktu konceptu do štandardu jazyka.

Zistili sme, že myšlienka zavedenia konceptov sa datuje k štandardu C++11, ale pre svoju zložitosť sa dostali až do štandardu C++20. Námetom na pokračovanie v tejto oblasti môže byť preskúmanie funkcionalít, ktoré boli pôvodnej verzii konceptov odobrané (ktoré sa nedostali do budúcich štandardov jazyka, ako napríklad konceptové mapy).

Ukážkou príkladov a popisom spôsobov viazania konceptov na parametrizované typy sme ukázali, že sú, vo väčšine prípadov, jednoduchším, čitateľsky prehľadnejším a precíznejším vyjadrením podmienok na dátové typy. Dôvodom prečo iba vo väčšine je ten, že sú, okrem tvorenia vlastných konceptov pomocou kľúčového slova `requires`, písané pomocou typových charakteristík, ktoré boli základom pre písanie podmienok s konštruktom `enable_if`.

Preskúmali sme, akým spôsobom sa diagnostikujú chybové výpisy kompilátora, a že majú určitý formát. Ďalším námetom na pokračovanie v oblasti môže byť preskúmanie interného fungovania kompilátora a jeho správania sa pri príchode ku konceptu.

Preštudovali sme preddefinované koncepty z knižnice `concepts`. Pri popise vybraných konceptov sme si uvedomili, že predpis konceptu je silne logický svojou syntaxou, pripomínajúc matematické definície. Práca bola prehľadového charakteru, ale námetom na pokračovanie v oblasti konceptov aj po praktickej stránke môže byť pokus o spísanie vlastných, zmysluplných a využiteľných konceptov, i keď zložitosť písania vecných a komplexných konceptov môže byť vysoká (ako sme uviedli v usmerneniach na písanie konceptov v kapitole 3).

V závere kapitoly 2 sme napísali podkapitolu 2.8 o porovnaní konceptov a spôsobov vymedzovania typu z kapitoly 1 a ukázali sme, že koncepty sú naozaj plnohodnotnou náhradou doterajších spôsobov s podobným zameraním.

Na záver práce sme si naštudovali a v práci spísali niektoré z usmernení k písaniu konceptov tak, aby koncepty spĺňali to, na čo boli do štandardu uvedené – vylepšenie čitateľnosti, vecnosti a správnosti.

Pôvodne sme mali v pláne si prejsť niektoré z projektov na štandarde C++20 s otvoreným zdrojovým kódom využívajúcich koncepty, a ktorých repozitáre sú priebežne aktualizované a udržiavané. Zistili sme, že takých verejných projektov je málo (alebo boli ťažko dohľadateľné, pretože neboli vhodne označené), a z toho väčšina bola edukačného charakteru z čias vydania štandardu C++20. Projekty využívajúce koncepty využívali jednoduché koncepty, zväčša kombinácie knižničných konceptov – usúdili sme, že nie sú vhodné na písanie ďalšej kapitoly.

Literatúra

[1] Concepts (C++) - Compiler Diagnostics. Dostupné z https://en.wikipedia.org/wiki/Concepts_(C%2B%2B)#Compiler_diagnostics, [Citované dňa 18/05/2024].

[2] Constraints and concepts - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/constraints, [Citované dňa 08/12/2023].

[3] Dokumentácia ku knižnici concepts. Dostupné z https://en.cppreference.com/w/cpp/concepts, [Citované dňa 23/04/2024].

[4] Funkcia min z Štandardnej šablónovej knižnice. Dostupné zo https://cplusplus.com/reference/algorithm/min/, [Citované dňa 01/03/2024].

[5] ISO C++, C++0x Concepts - Historical FAQs. Dostupné z https://isocpp.org/wiki/faq/cpp0x-concepts-history, [Citované dňa 22/01/2024].

[6] Nástroj Doxygen. Dostupné z https://doxygen.nl/manual/docblocks.html.

[7] Operand typeid - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/typeid, [Citované dňa 15/02/2024].

[8] Placeholder type specifiers (auto) - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/auto, [Citované dňa 15/02/2024].

[9] SFINAE - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/sfinae, [Citované dňa 4/04/2024].

[10] Skrátená šablóna funkcie - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/function_template#Abbreviated_function_template, [Citované dňa 14/04/2024].

[11] static_assert declaration - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/static_assert, [Citované dňa 16/02/2024].

[12] std::convertible_to - cppreference. Dostupné z https://en.cppreference.com/w/cpp/concepts/convertible_to, [Citované dňa 13/05/2024].

[13] std::derived_from - cppreference. Dostupné z https://en.cppreference.com/w/cpp/concepts/derived_from, [Citované dňa 29/04/2024].

[14] std::enable_if - cppreference. Dostupné z https://en.cppreference.com/w/cpp/types/enable_if, [Citované dňa 5/04/2024].

[15] std::floating_point - cppreference. Dostupné z https://en.cppreference.com/w/cpp/concepts/floating_point, [Citované dňa 01/03/2024].

[16] std::integral - cppreference. Dostupné z https://en.cppreference.com/w/cpp/concepts/integral, [Citované dňa 01/03/2024].

[17] std::is_arithmetic. Dostupné z https://en.cppreference.com/w/cpp/types/is_arithmetic, [Citované dňa 16/02/2024].

[18] std::same_as - cppreference. Dostupné z https://en.cppreference.com/w/cpp/concepts/same_as, [Citované dňa 29/04/2024].

[19] Templates - cppreference. Dostupné z https://en.cppreference.com/w/cpp/language/templates, [Citované dňa 15/02/2024].

[20] type_traits header - cppreference. Dostupné z https://en.cppreference.com/w/cpp/header/type_traits, [Citované dňa 1/03/2024].

[21] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, pages 28–115, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. [Citované dňa 28/02/2024].

[22] Bartłomiej Filipek. Notes on C++ SFINAE, Modern C++ and C++20 Concepts, 2020. Dostupné z https://www.cppstories.com/2016/02/notes-on-c-sfinae/, [Citované dňa 4/04/2024].

[23] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, oct 2006. [Citované dňa 28/02/2024].

[24] Kent D. Lee. *Programming Languages: An Active Learning Approach*. Springer Science Business Media, 2008. [Citované dňa 07/03/2024].

[25] David R. Musser and Alexander A. Stepanov. Generic programming. In P. Gianni, editor, *Symbolic and Algebraic Computation*, pages 13–25, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. [Citované dňa 07/03/2024].

[26] František Silváši. Syntaktické a sémantické požiadavky na parametrizované typy v C++. Bakalárska práca, Technická univerzita v Košiciach, 2014. [Citované dňa 22/01/2024].

[27] Ellen Spertus. Best practices for writing code comments, 2022. Dostupné z `https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments`.

[28] Bjarne Stroustrup. The Future of Generic Programming or How to design good concepts and use them well. Technical Report P0557r1, Morgan Stanley and Columbia University, January 2017. [Citované dňa 12/04/2024].

[29] Bjarne Stroustrup and Herb Sutter. C++ Core guidelines. Dostupné z `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`, [Citované dňa 25/04/2024].

[30] Andrew Sutton and Bjarne Stroustrup. std::sortable - cppreference. Dostupné z `https://en.cppreference.com/w/cpp/iterator/sortable`, [Citované dňa 03/04/2024].

[31] Šimon Tóth. C++20 Concepts - Complete Guide, 2021. Dostupné z `https://itnext.io/c-20-concepts-complete-guide-42c9e009c6bf`, [Citované dňa 19/04/2024].

[32] Bruno van Dooren. An Introduction to C++ Concepts for Template Specialization, 2022. Dostupné z `https://www.codeproject.com/Articles/5340890/An-Introduction-to-Cplusplus-Concepts-for-Template`, [Citované dňa 19/04/2024].

[33] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, 2002.