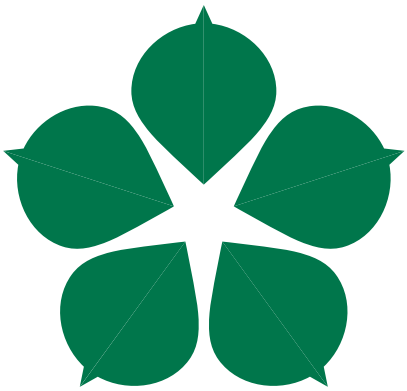
Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta



Moderní programování v jazyce C++ a jeho efektivita

Bakalářská práce

Tomáš Zeman

Vedoucí práce: Ing. Jan Fesl, Ph.D.

České Budějovice 2022

Přírodovědecká fakulta
Faculty of Science

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

ZADÁVACÍ PROTOKOL BAKALÁŘSKÉ/~~DIPLOMOVÉ~~[1] PRÁCE

Student: Tomáš Zeman
(jméno, příjmení, tituly)

Program studia/specializace: Aplikovaná informatika - Web a multimédia

Pracoviště PřF JU, kde bude práce vypracována a obhájena: Katedra informatiky

Školitel: Ing. Jan Fesl, Ph.D.
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, e-mail)

Garant z PřF:
(jméno, příjmení, tituly, katedra – jen v případě externího školitele)

Školitel – specialista, konzultant:
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, e-mail)

Téma práce: Moderní programování v jazyce C++ a jeho efektivita

Cíle práce:

Cílem práce je pochopení a osvojení si dovedností vycházejících z detailních poznatků moderní verze jazyka C++ (11/14/17/20) a reálné otestování efektivity různých programovacích konstrukcí (srovnáním s předchozími implementacemi). Student se zaměří zejména na operační složitost použitých implementací vybraných algoritmů, popř. datových struktur a provede jejich srovnání. Srovnání budou podložena průkaznými měřeními a vyhodnocena vhodnými statistickými metodami.

V další fázi se student zaměří na vliv konkrétního překladače, resp. některé z jeho částí (např. optimalizátor či generátor kódu) s ohledem na efektivitu binárních forem zkompilovaných programů . Student navrhne a provede praktická měření pro srovnání efektivity v současnosti nejběžněji používaných překladačů (g++, clang a MS Visual Studio compiler) a pokusí se vysvětlit eventuální rozdíly mezi nimi.

Pro zajištění co nejvyšší rigoróznosti výsledků bude srovnání efektivity provedeno na reprezentativní množině vybraných programů zahrnující statisticky významný počet programovacích konstrukcí resp. algoritmů.

---

[1] Nehodící se škrtněte/smažte

Základní doporučená literatura:

MEYERS, Scott. Effective modern C++: 42 specific ways to improve your use of C++11 and C++14. Sebastopol, CA: O'Reilly Media, 2014. ISBN 1491903996.

ALEXANDRESCU, Andrei. Moderní programování v C++: návrhové vzory a generické programování v praxi. Brno: Computer Press, 2004. ISBN 80-251-0370.

CALANDRA, Anthony. Modern C++ features, dostupné online - https://github.com/AnthonyCalandra/modern-cpp-features.

Zvláštní poznámky pracoviště:

Financování práce:

Školitel práce	podpis : .....
U externích vedoucích fakultní garant práce	podpis : .....
Garant programu/specializace[2][3]	podpis : .....
Vedoucí pracoviště PřF JU, kde proběhne obhajoba	podpis : .....
Souhlas vedoucího ústavu AV nebo jiné instituce[3]	podpis : .....

V Českých Budějovicích dne 01.09.2021

podpis studenta: .....

---

[2] v případě prací v bakalářském programu Biologie není podpis garanta programu vyžadován [3] v případě magisterských prací v programech učitelství pro SŠ podpis proděkana pro učitelské obory

[3] v případě, že práce bude vypracovávána jinde než prostorách PřF, například na ústavu AV

Bibliografické údaje

Zeman, T., 2022: Moderní programování v jazyce C++ a jeho efektivita. [Modern C++ programming and its efficiency. Bc. Thesis, in Czech.] – 88 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic

Annotation

The aim of bachelor thesis is to use effective techniques and skills of modern C++ programming. The stated goal is to compare the latest standard and its features in terms of improved performance and its efficiency. Thus, the thesis includes an in-depth testing of selected features and data structures, which are then compared and evaluated according to complete statistical methods. In the testing phase, the work also targets on the most used compilers today (g++, Clang and MS Visual Studio compiler), or their impact in terms of the resulting efficiency of the generated executable form. The analysis thus aims at achieving new, useful results and the appropriate presentation of their differences.

Keywords:

C++ language, performance testing, efficiency in programming, compiler, ISO C++ standard

Prohlašuji, že svoji [bakalářskou – diplomovou] práci jsem vypracoval/a samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své [bakalářské – diplomové] práce, a to [v nezkrácené podobě – v úpravě vzniklé vypuštěním vyznačených částí archivovaných Přírodovědeckou fakultou] elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 12.11.2022 Podpis autora 

Poděkování

Rád bych touto cestou poděkoval vedoucímu mé bakalářské práce Ing. Janu Feslovi, Ph.D. za vedení, odbornou pomoc, konzultace a vstřícnost při vypracování této práce.

Obsah

Úvod

V současné době je jazyk C++ stále jedním z předních objektově orientovaných jazyků. Od jeho vzniku z 80. let 20.století prošel řadou změn a vylepšení. Psaní programů se poté zejména od standardu C++11, jenž přinesl doslova revoluční změny, značně zefektivnilo. Standard přinesl určité usnadnění a spoustu moderních konstrukcí, přičemž se mu povedlo do jisté míry zachovat kompatibilitu se standardem přechozím. Podstatnou otázkou tak může být, co je moderní C++. Samotná odpověď na ni nemusí být úplně jednoznačná. Pojem jako takový můžeme primárně definovat jako psaní efektivních a rychlých programů, které využívají funkcionalit z posledních standardů včetně stálého udržování jednoduchosti a přehlednosti kódu. Dobře napsané programy jsou díky tomu hardwarově optimalizované, úspornější a méně náchylné k chybám.

Práce se zpočátku zabývá historickým kontextem samotného jazyka a jeho standardizací C++ (11/14/17), přičemž je uváděno několik zajímavých konstrukcí, které do jazyka přinesly výrazné změny, a to především z hlediska samotné efektivity a rychlosti. V práci bude dále pasáž zatím posledního standardu C++20, který bude detailněji rozebrán včetně praktického otestování nových funkcionalit, jejich zhodnocení a shrnutí daného přínosu.

Standard C++11 přinesl v historickém měřítku dramatické změny oproti C++98. Převrat přinesl to, že jazyk se stal více expresivním, snazším na používání a dosahoval daleko lepšího výkonu v aplikacích. Nejnovější standard C++20 samotný autor jazyka označil[1] za skokové zlepšení a v určitých ohledech i obdobné zvelebení, jako tomu bylo kdysi u vydání verze C++11. Pro vývojáře tak dnes jazyk nabízí pevnou stabilitu, spolehlivost, přenositelnost a bezkonkurenční výkon. Objektový přístup jazyka také přináší snadné přenesení problémů reálného světa. Postupné vylepšování jazyka chápeme v drobných evolučních změnách, jako je zachování jednoduchosti při používání, progresivní rozvoj generického programování, vylepšení vícevláknových aplikací a výpočtů v době překladu, ale taktéž poskytnutí určité podpory staršímu kódu. [1]

Jazyk C++ má oproti ostatním konkurentům signifikantní převahu v oblastech, kde je potřeba s hardwarem nakládat úsporně a efektivně, což mohou být procesy se značnou výpočetní složitostí. Jazyk byl totiž v samotném principu navržen pro přímou komunikaci s používaným hardwarem a zároveň maximálně podpořen v jeho abstrakci.

[1] Více informací je v deváté kapitole zdroje [1]

Výzkum [2] z roku 2017 analyzuje efektivitu u 27 nejčastěji používanými programovacími jazyky, přičemž právě jazyk C/C++ dosahuje ve všech provedených testech pokaždé na přední příčky.

Stále se však jedná o nástroj, který se v čase mění s novými potřebami a uzpůsobuje se tak novým výzvám. Jeho dnešní uplatnění je například ve vědě, automobilové technice, embedded systémech, medicíně a biologii, průmyslu, bankovních a telekomunikačních službách, operačních systémech, cloudových službách a mnoha dalších. [1]

V současné době se jazyk C++ markantně vyvíjí, přičemž stále poslední standard zůstává relativně neprobádaný a je potřeba přijít s novými poznatky do oblasti moderního programování. Využití efektivních programovacích technik může vést ke značenému zlepšení softwarových aplikací a k ušetření nemalých prostředků.

Stanoveným cílem práce je porovnání standardů na testovaných algoritmech vzhledem k tomu, zda a případně jaké posunutí nastalo, a následné předložení podrobných výsledků zhodnocující efektivitu a přínos.

Nejen implementovaný kód a moderní konstrukce však mohou mít vliv na výsledný výkon programu, ale taktéž vybraný překladač. Současně také v testovací fázi proběhne analýza překladačů a jejich statistické vyhodnocení, zda generují stejně efektivní spustitelnou formu kódu.

1 Evoluce jazyka C++

1.1 Historický kontext

Na počátku roku 1979 pracuje Bjarne Stroustrup, v té době ještě jako student, na své disertační práci v laboratořích na univerzitě v Cambridge. Jeho cílem tehdy bylo prostudování a naprogramování událostmi řízené simulace, jež by byly přímo navrženy pro distribuované systémy. Počáteční verze takového programu byla již napsána v jazyce Simula 67, přičemž právě tento jazyk je považován za vůbec první funkční objektový jazyk, jenž také jako první uvedl klíčové slovo třída. Takovýto objektový přístup byl tehdy vzhledem k jeho vlastnostem pro simulátor ideální. Přístup dovoloval mapování konceptů do konstrukcí, které tak byly prozatím u jiných jazyků nevídané a díky tomu kód působil daleko přehlednějším a čitelnějším dojmem. Vlastnosti jako dědičnost a hierarchie tříd pak usnadňovaly snadné rozšiřování a organizaci celého kódu. Důležitou součástí však byl i samotný kompilátor jazyka, který byl silně typový a chyby uměl zachytil během překladu. Tato nová objektová ideologie k přístupu u programování byla pak prvotní inspirací pro budoucí vývoj jazyka C++.

Tím, jak projekt v čase narůstal, se více začala projevovat výhoda nového přístupu a to, že se software začal chápat a vyvíjet jako kolekce menších podprogramů než jen jako jeden ohromný celek. Celý objektový přístup se tak ukázal jako průlomový a je dodnes stále populární. Pojetí přináší do světa programování výhody opětovné použitelnosti, snadného vytváření knihoven, přehlednosti kódu, a také i jeho samotné odladění.

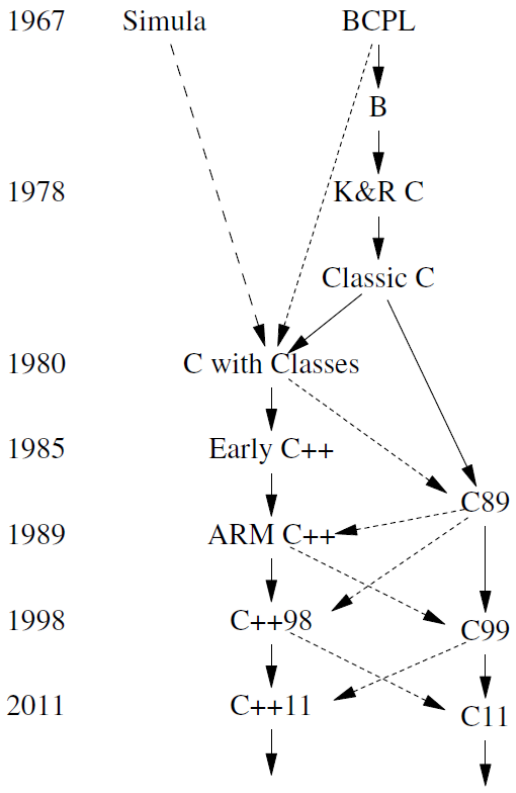
Značnou nevýhodou jazyku Simula 67 byly v té době až propastně dlouhé časy překládání a bohužel i pomalá doba běhu programu. Jazyk byl v té době vhodný spíše pro malé projekty, ale prakticky nepoužitelný u těch rozsáhlejších. Za pomalou dobou běhu však mohly spíše chybné charakteristiky při návrhu jazyka než samotná funkcionalita vytvořené aplikace. Jazyku navíc chyběla možnost přetěžování operátorů a další dnes již běžné vlastnosti.

Stroustrup tak byl nucen kompletně přepsat celý projekt, a to do strukturovaného jazyka BCPL, který však působil tehdy vedle jazyka C velmi jednoduchým dojmem. Jeho nevýhodou bylo rovněž neposkytování žádné typové kontroly při překládání. Zdařilým výsledem ale bylo to, že simulace dosahovala uspokojivých běhových časů a poskytla užitečné výstupy. [3]

Tyto pozitivní i negativní zkušenosti tak vedly autora jazyka C++ k vytvoření nového přelomového programovacího jazyka.

1.1.1 Základní pilíře efektivního jazyka

1. Organizace samotného programu podobně, jako ho využíval jazyk Simula 67. To znamená objektový přístup, hierarchičnost, silná typová kontrola a možnost vytváření koprogramů.
2. Rychlá doba běhu a snadné zkombinování jednotlivě překládaných částí pro jeden program. Opakovatelná použitelnost kódu, napsané komponenty tak snadněji zařazovat do programu.
3. Nástroj, jehož výsledná implementace bude moci být přenositelná a dostupná i pro tehdy naprosto běžné počítače. Výsledný program pak nesmí být zbytečně hardwarově zatěžující a musí se s ním efektivně a napřímo komunikovat. [3]



Obrázek 1 Evoluce jazyka do roku 2011 [4]

1.2 C with Classes

Po stanovení kritérií započala práce a Stroustrup deklaruje významný cíl. Vytvoření programovacího jazyka, který by dokázal maximálně naplňovat předchozí podmínky a jako vhodným základem byl pro něj jazyk C. Projekt nese prozatímní název „C with Classes“, označující tak stěžejní objektovou stránku jazyka. Samotná organizace projektu nebyla nijak centrálně řízena. Projekt tak čistě vzniknul na základě aktuálních potřeb. [3] [5]

Celou motivaci autor dle vlastních slov shrnuje takto: „Jazyk C++ byl primárně navržen tak, abychom já a mí přátelé nemuseli psát programy v assembleru nebo v tehdy aktuálních vyšších jazycích. Jeho hlavním účelem bylo zpříjemnění a zjednodušení psaní dobrých programů všem programátorům.“ [přímá citace, volně přeloženo z [6], strana 23]

1.2.1 Kompatibilita s jazykem C

Samotná kompatibilita byla téměř plně zachována, avšak v té době nebyla nijak vynucována. Na druhou stranu tím, že byla téměř úplně udržena, usnadnila tak valné většině programátorů přechod z jazyka C do C++. [7]

Požadovaný jazyk měl být všestranně dostupný s obecným mechanismem pro organizaci programů. Zde ukazují svou neodmyslitelnou výhodu knihovny, které tyto specifické potřeby dokáží zajistit. Knihovny se dají definovat jako kolekce programových modulů, které můžeme přidávat do programů. Poskytují kolekci mnoha vyřešených a odladěných programových problémů, které programátorům šetří čas. [8]

1.2.2 Jazyk C jako výchozí řešení

- **Flexibilita**: Volba jazyka C byla v té době vhodná téměř do všech systémových úloh. Jazyk totiž nemá v sobě vrozené limitace použití pro specifickou oblast v programování. Vývojáři tak nebyli omezeni výběrem toho jazyka.
- **Výkonnost**: Sémantika jazyka C je nízko-úrovňová. Pro programátora a pro překladač je tak relativně snadné zužitkovat hardwarové zdroje pro napsaný program.
- **Dostupnost**: Znamená použití v té době široce variabilního množství počítačů. Od těch méně výkonných až po superpočítače. Přístup ke knihovnám pro vývojáře, než aby museli opakovat stejný design a konstrukce psát od začátku. Jazyk C má

charakteristiky stručného, přehledného jazyka s velkou dostupností a vazbou na operační systém UNIX.

- **Přenositelnost**: Program napsaný v C není automaticky přenositelný z jednoho počítače na druhý, a to včetně operačních systémů. Přenesení důležitých částí softwaru je však ale stále ekonomicky přijatelné. [3]

Vývoj jazyka C with Classes probíhal ve výzkumných laboratořích AT&T Bell Labs. Mnoho zajímavých myšlenek ale bylo zahozeno z různých důvodů. Jednalo se například o velký limitační nárok a použití v reálných projektech, velmi těžko implementované řešení, příliš paměťově nebo časově neefektivní, nebo by vznikla až příliš velká nekompatibilita s jazykem C. [3]

1.2.3 Přínos jazyka C with Classes

Jazyk v roce 1980 přináší: třídy a odvozené třídy, konstruktor a destruktor, přístup typu public a private, typovou kontrolu a konverzi u argumentů funkcí. Později při přejmenovaní na jazyk C++ se objevily stěžejní vlastnosti: virtuální funkce, přetěžování operátorů, reference, konstanty a vylepšení typové kontroly. [3]

- Úspěch jazyka sám autor v té době označuje za ucházející.
- Jazyk dokázal značně zlepšit vývoj primárně u větších programů, aniž by došlo ke ztrátě efektivity při samotném běhu. Druhým aspektem bylo to, že se jazyk nevychyloval od jazyka C a snadno se tak na něj přecházelo. To je významné hledisko u velkých společností, které by změny neakceptovaly.
- Stroustrup požaduje pro jazyk nové pojmenování [2]. Zapotřebí je nicméně nový a modernější překladač. [3]

1.3 Cfront

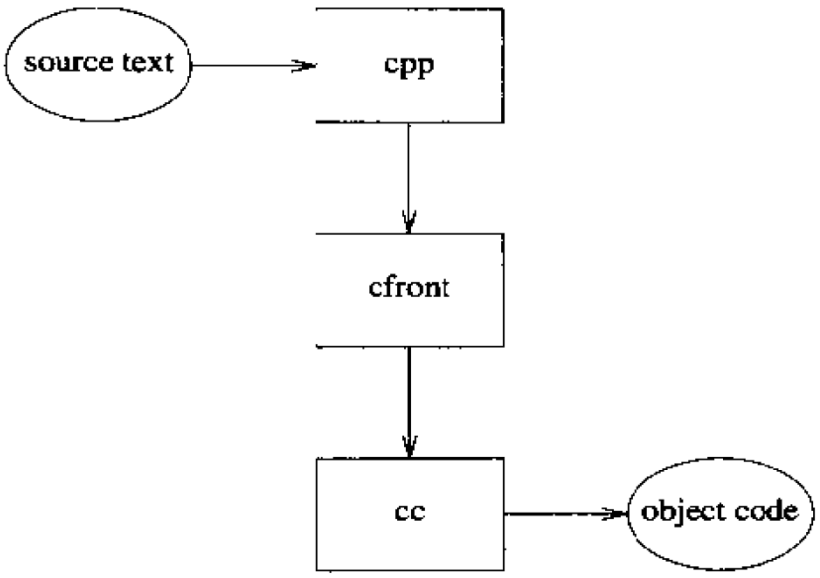
Překladač Cfront byl poprvé navržen (a napsán v jazyce C++) Stroustrupem v roce 1982, přitom samotná implementace trvala více než rok, a tak v říjnu roku 1983 byl poprvé veřejně dostupný pro uživatele. Cfront je tradičním překladačem, používající lexikální a syntaktický analyzátor, přičemž kompilátor uměl kompletně vytvořit vnitřní stromovou reprezentaci tříd, funkcí a dalších bloků. Rovněž bylo zapotřebí provést určitou

[2] Nejprve zváženo C84, poté v roce 1983 C++, autorem je Rick Mascitti [3]

optimalizaci na úrovni zdroje daných C++ konstrukcí před tím, než by se dostaly na samotný výstup. [1]

Zásluhou všech těchto aspektů dosahoval překladač rychlých časů překladu a dle autora [3] i obdivuhodně rychlého běhu programu. Překladač principiálně fungoval tak, že program napsaný v C++ byl nejprve přeložen do jazyka C a ten následně do strojového kódu. Další verze tohoto překladače pak později přinesly rozšíření v podobě přetěžování operátorů, vícenásobného dědění a šablon. [7]

Důvod, proč bylo nezbytné vygenerovat výstup v jazyce C, byl pádný. V té době bylo totiž nesčetné množství nestandardizovaných linkerů a optimalizátorů, než je tomu nyní v současnosti, a výběr jazyka C tak bylo na místě. Kromě toho byl také kladen určitý požadavek na výstup, na to, aby byl co možná nejsnadněji přenositelný. Tyto skutečnosti tak poukazují na bezesporný fakt, že assembler u jazyka C byl tehdy prostě tím nejvhodnějším nástrojem. [3]



Obrázek 2 Schéma překladače Cfront [3]

2 Jazyk C++

Oficiální vydání jazyka C++ bylo v roce 1985 a přineslo tak velmi praktický a užitečný nástroj nejen pro profesionální vývojáře. Jeho hlavním dopadem bylo poskytnutí průlomového jazyka, než jsou v té době dosavadní, podporu na úrovni datové abstrakce a objektově orientovaného přístupu. Účelem jazyka C bylo nahrazení dosavadního programování v assembleru pro systémové úlohy velmi náročných kvalit. Jazyk C++ v tomto ohledu nebyl při návrhu nijak omezen a lze ho stále používat pro optimalizování úloh na velmi nízké úrovni. [5]

2.1 Revoluční nástroj

Autorovi se povedlo vytvořit moderní jazyk na základech jazyka C, aniž by nějak zásadně modifikoval jeho strukturu. Tím se až na určité výjimky zachovala kompatibilita. Jazyk C++ [3] je tak nadstavbou jazyka C, který je jeho podmnožinou. Primárním rozdílem je, že C je jazyk strukturovaný a C++ objektový. [3]

Evoluce a rozvoj samotného jazyka musí být nadále řízen dle aktuálních potřeb. Je nezbytné sledovat to, jak se reálné projekty vyvíjí a poskytují tak na úrovni jazyka potřebnou optimalizaci. Přenositelnost a výkon napsaných aplikací je tak důležitějším aspektem, než aby v nesprávných místech docházelo k jazykové simplifikaci. Vylepšování je potřeba uzpůsobovat tak, aby si jazyk poradil se stupňováním obtížnosti v moderních programových výzvách. [3]

2.1.1 Charakteristika jazyka

C++ není pouhým rozšířením o objektovou stránku jazyka C, ale zcela novým programovacím jazykem přebírající stěžejní množinu syntaktických i sémantických prvků. Řada jazyků jako je Java, Javascript, PHP a C# z něj vychází, avšak zde bylo mnoho konceptů oproti C++ zjednodušeno. [5]

2.1.2 Vlastnosti

- Imperativní, objektově orientovaný jazyk.
- Široké rozšíření a přenositelnost programů mezi operačními systémy.
- Podpora optimalizovaných knihoven.

[3] Název odkazuje na inkrementační operátor.

- Staticky[4] silně typový, podporující generické a meta programování s pomocí šablon. [9]
- Vysoká úroveň výkonu a efektivní práce s pamětí, unikátní pohled na sémantiku objektů oproti ostatním jazykům a používání ukazatelů.
- Ideální pro rozsáhlé projekty, operační a embedded systémy, systémové a mobilní aplikace, knihovny a překladače pro jiné programovací jazyky. [10]

2.2 Objektově orientované programování

Od strukturovaného programování, které klade především důraz na využívání algoritmů, objektové zdůrazňuje data. Hlavní myšlenkou objektového přístupu je praktické používání datový tříd, které se nejvíce přibližují reálným potřebám. Samotná instance třídy je pak určitá datová struktura, která je ze třídy vykonstruovaná[5]. Třída obecně definuje všechny potřebné údaje, které se použijí k reprezentaci objektu a jeho činností, které může vykonávat. Praktickou výhodou OOP je znovu použitelnost již odladěného kódu.

2.2.1 Šablony

Po určité době popularity samotného jazyka rozšířil autor jazyk o programovaní za pomocí šablon. Tento prvek se ukázal pro jazyk klíčový a je považován za dosud nejdůležitěji dodatek pro C++. Skutečnost, že jazyk používá šablony, ukazuje na to, že C++ vede důraz na univerzálnost jazyka jako takového, což je pravděpodobně jeden z hlavních faktorů celosvětového úspěchu jazyka. [5] [8]

2.2.2 Aspekty OOP

Zapouzdření

Znamená použití strukturovaných datových typů pro reálnou reprezentaci daných objektů. Třída jako taková obsahuje atributy a metody, což jsou konkrétní předpisy pro manipulaci s objekty. Uzavření části kódu programu do jediného datového typu je vedeno tak, že tvoří atomický celek. Zapouzdření taktéž definujeme jako schování dané implementace do jednoho logického celku.

[4] Typová kontrola se provádí při překladu na rozdíl od jazyků, které ho provádějí při běhu.
[5] Odsud pojem konstruktor sloužící k inicializaci objektu, opakem je pak destruktor.

Dědičnost

Umožnuje znovupoužití již vytvořených tříd. Nově vytvořené typy objektů lze tak definovat na základě dříve hotových. Všechny atributy a metody se pak dědí z předka na potomka, přičemž C++ umožnuje vícenásobnou dědičnost. Novou třídu můžeme poté specializovat přidáním nových atributů a metod.

Polymorfismus

U více objektů stejného typu lze definovat vlastní metody a operátory, které při zavolání mohou provádět různě specifické činnosti. K docílení polymorfismu za běhu programu se pak používají virtuální metody a třídu tak nazýváme jako polymorfickou. [5]

2.3 Standardizace jazyka

Standard byl z historického hlediska pro jazyk C++ velkým milníkem. Neboť jasně definuje přesný obsah a chování jazyka. Daný jazyk je méně komplikovaný při programování, snadněji se vyučuje na univerzitách a dovoluje přenesení programů mezi jednotlivými platformami. Standard tak významně pomáhá programátorům ve stavění jejich aplikací, které dosahují lepších výsledků s úspornějším kódem. [11]

Standard definujeme jako specifikaci, nikoliv implementaci. Jeho účelem je pak udržovat více implementací v určité dohodě. Je potřeba rozhodnout, co dohoda ve skutečnosti znamená ve světě rozmanitého hardwaru, tak aby s ním bylo nakládáno co nejefektivněji. [1]

2.3.1 WG21

Komise určená k samotnému rozvoji ISO C++ standardu se nazývá WG21. Od roku 1990 je tato komise hlavním evolučním vlivem jazyka C++ a zajišťuje jeho uhlazenost a přesnou definici.

Standard nám říká, co a jak v jazyku funguje. Nenařizuje nám však pravidla, jak jej správně a efektivně využívat. Zde jsou signifikantní rozdíly v pochopení a porozumění technickým detailům v programování a mezi tím, jak efektivně jazyk kombinovat s dalšími vlastnostmi, nástroji a knihovnami k vyprodukování lepšího softwaru. Ten zde v tomto případě má význam přenositelnosti, odolnosti vůči chybám, ale i rychlosti. Vydání standardů tak má naplňovat jistou očekávanost a přinesení zásadních funkcionalit pro jazyk, které se s postupem technických možností objevují. Tříletý plán je tak do jisté

míry ambiciózní myšlenka s tím, že standardy střídají fáze předního vydání a drobná vylepšení. [12]

2.3.2 ISO C++ standard

Z historického hlediska se v roce 1989 za první oficiální standard jazyka C považuje ANSI dokument pod označením X3J11/90-013. O rok později však byl přijat mezinárodní standard ISO označovaný jako C90 a je dnes stále běžně používán. Později v roce 1999 byla pro jazyk vydána verze s označením C99. Ta především do jazyka zahrnula nejvíce požadovaná rozšíření včetně oprav potencionálních chyb. Pro jazyk C++ jako takový byl přijat první standard v roce 1998, přičemž tento dokument se označuje ISO/IEC 14882-1998. [7]

Následující tabulka zobrazuje přehled vydaných standardů a jejich přínosu do jazyka C++ od roku 1998 do současnosti.

Tabulka 1 Standardizace jazyka

Standardizace jazyka v průběhu let		
Rok	Standard C++	Hlavní přínos
1998	C++98, ISO/IEC 14882:1998	STL algoritmy, knihovna iostream v std, datový typ bool.
2003	C++03, ISO/IEC 14882:2003	Přínos spíše ve smyslu oprav drobných a technických chyb, nikoliv však podstatných změn.
2011	C++11, ISO/IEC 14882:2011	Auto type, lambda funkce, move sémantika, nullptr, range-based cykly, variadické šablony, constexpr a mnoho dalšího.
2014	C++14, ISO/IEC 14882:2014	Genericita u lambda funkcí, std::make_unique, použití typu auto pro návratový typ u funkcí.
2017	C++17, ISO/IEC 14882:2017	Přidání constexpr u lambda funkcí, zanořování jmenných prostorů, fold expressions.
2020	C++20, ISO/IEC 14882:2020	Moduly, optimalizační atributy likely a unlikely, coroutines, immediate funkce, constexpr virtuálních funkcí, concepts, a další.

Výchozí zdroje pro tabulku [13] [14]

3 C++11/14/17

V této kapitole budou vysvětleny vybrané, a především zásadní vlastnosti ze standardu C++11/14/17, které výrazně v pozitivním smyslu ovlivňují používání daného jazyka. Tyto vybrané konstrukce a nová klíčová slova především zlepšují aplikace nejen z hlediska efektivity, ale také v přehlednosti, generickém programování, silnějších kontrolách, jež přinášejí všem programátorům výhodnější řešení.

3.1 Návrhové cíle pro zlepšení jazyka C++11

- Nadále zdokonalovat jazyk v systémovém programování a rozšíření jeho standardní knihovny.
- Podnítit dosavadní jazyk v jeho unikátních přednostech než zabíhat do oblastí, kde má slabé stránky. Nepodporovat či nezavádět jednoúčelové zaměření.
- Distribuovat jazyk veřejnosti ve větší dostupnosti, obzvláště pak pro jeho vyučování na univerzitách i k cílenému samostudiu. Udržovat vlastnost, aby jazyk byl pro nové i stávající programátory pochopitelný a následné znalosti byly do praxe snadno přenositelné.
- Zachování stability a kompatibility se staršími verzemi jazyka C a C++ jak to jen bude možné. Zaměřit se na zdokonalování schopností pro přímou komunikaci s hardwarem, zlepšení výpočetního výkonu u embedded systémů a výpočetních clusterů. [15]

3.2 C++14

Standard C++14 byl kladen s cílem na vylepšení a doladění předchozí verze. Významný či úplně nový přínos tak zde očekáván nebyl. Komise WG21 však dokázala, že standard je schopna dodat opět včas v rámci tříletého cyklu. K nemalému posunu došlo zejména u constexpr a lambda funkcí, které jsou nově rozšířeny oproti dosavadnímu používání. Přechod z C++11 na C++14 byl relativně bezproblémový a nezpůsoboval zásadní chybu na nízko-úrovňovém rozhraní [6], jež dovoluje již přeloženému programu jeho funkčnost na všech dosavadních systémech, aniž by se do něj zasahovalo. [1]

[6] Tzv. Application Binary Interface

3.3 C++17

Po předchozím standardu C++14 se zde původně mělo jednat o zásadní vydání. Očekávané vlastnosti tak měly pozměnit způsob dosavadního vytváření softwaru. Ve výsledku se však povedlo naplnit a obohatit jazyk o spoustu drobných rozšíření, avšak nejvíce očekávané vlastnosti jazyka, jako jsou concepts, coroutines a modules, zde zatím stále chyběly. Ty byly nicméně dodány později až s vydáním standardu C++20. Ve výsledku tak standard C++17 přinesl důležité změny, ve kterých si valná většina vývojářů našla určité kvality. V tomto případě se bohužel nejednalo o nic tak zásadního, co by doposud změnilo koncept vyvíjení softwaru. Posouvat tak evolučně jazyk dál a zároveň ho udržet jednoduchý na používání je stále značná a pro komisi zásadní výzva. [12]

3.4 Auto type

Klíčové slovo „auto“ je velmi užitečný způsob, jak deklarovat proměnnou, která má složitý datový typ. Překladači tak napovídá, aby typ sám odvodil z daného objektu. Toto klíčové slovo je prospěšné zejména v situacích při používání šablon, ukazatelů, u lambda funkcí a v cyklech. [16]

Auto typ z pravidla používáme za okolností, kdy nepotřebujeme (kromě unikátních případů) explicitně vyjádřit datový typ. Specifickým případem může být, požadujeme-li výslovné vyjádření přesného rozsahu proměnné (preferování double před float), z čehož se také vyvozuje daná viditelnost datového typu. Použitím klíčového slova „auto“ se ale vyhýbáme chybám z pozdějších úprav, refundaci v kódu a dlouhých názvů u datových typů. [4]

Vypsání hodnot z pole s pomocí auto typu.

```
for(const auto &x : arr)
    std::cout << x << std::endl;
```

Proměnná představující pole zde může reprezentovat například statické pole, vektor nebo list s různými datovými typy: integer, string, char, float a podobně.

Dalším efektivním využitím může být následující situace:

```
std::vector<int> v;
auto v_size = v.size();
```

Tím, že explicitně nedeklarujeme datový typ (např. unsigned), straníme se problémům u 64bitových operačních systémů Windows. Zde jsou totiž pro datový typ unsigned a std::vector<int >:: size_type dva různé rozsahy. Použitím auto typu se tak vyhneme neočekávanému chování a předejdeme kódovému zásahu při přechodu mezi platformami. [17]

Vhodným zvolením klíčového slova auto namísto std::function, jenž může nést jakýkoliv typ volaného objektu, dosáhneme úspory v paměti. Auto deklarovaná proměnná spotřebuje jen tolik paměti, kolik si konstrukce sama vyžádá. U std::function si ale může konstruktor alokovat neadekvátně velké množství paměti. Samotné volání objektu pak bude rovněž pomalejší než přes auto typ. [17]

decltype(auto)

U C++14 došlo k pokročilému vylepšení, klíčové slovo decltype(auto) při deklaraci proměnné nejen dedukuje její datový typ, ale zachovává také reference, volatile a konstanty. [14]

```
int x = 0;
int& x1 = x;
auto x2 = x1;
decltype(auto) x3 = x1;
std::cout<<&x1<<“ “<<&x2<<“ “<<&x3<<std::endl; // x1, x3 mají shodné adresy
```

Return type deduction nově také umožnuje u funkcí použít klíčové slovo jako návratový typ, jenž tak zajistí genericitu kódu.

```
template <class T, class V>
auto f(T x, V y)
{
    return x > y ? x : y;
}
```

3.4.1 Výhody používání auto typu

- **Výkon:** auto garantuje, že nedojde k žádné konverzi.
- **Správnost:** zaručení toho, že při použití dostaneme adekvátní datový typ, odstraníme redundanci a zachováme přehlednost.
- **Reakce na změny:** kód je odolnější vůči budoucím zásahům a změnám. [16]

3.5 Constexpr

Definováním klíčového slova constexpr značíme určitou evaluaci v době překladu. Aby samotný výpočet mohl být proveden v kompilační době, musí funkci nebo proměnné předcházet klíčové slovo constexpr. [4]

Toto klíčové slovo bylo přidáno do standardu C++11 a jeho hlavní ideou je zlepšení výkonnosti programů za pomocí výpočtů při samotném překladu. Důležitý výpočetní výkon tak raději bude evaluován při kompilaci programu, a tím ho naopak pošetříme při běhu.

V C++11 nám dovolovala constexpr funkce jen velmi základní, ne příliš komplexní používání. Později u C++14 došlo k markantnímu rozšíření z hlediska použitelnosti tohoto klíčového slova. Funkcím totiž umožnil větvení v kódu, používání cyklů a vracení výsledků z více návratových hodnot.

Pokud se funkce zavolá s parametry, které však nejsou známy v době překladu, pak se zachová standardně a výpočet provádí až při běhu programu. Je-li však hodnota vypočtena během překladu, program poběží rychleji a bude paměťově úspornější. Tato vlastnost je pro programátory výhodná a nepotřebují tak nutně deklarovat dvě významově stejné funkce. Uvedenou definici však provázely až do C++20 i určité nevýhody. [17] [18]

Použití constexpr funkce vracející booleovskou hodnotu na porovnání dvou čísel. [19]

```
constexpr bool isGreater(int x, int y);

void foo(int v)
{
  bool a = isGreater(-5, 10); // evaluace nejspíše v době kompilace
  constexpr bool b = isGreater(-1, 2); // jistá evaluace v době kompilace
  bool c = isGreater (-1, v); // jistá evaluace v době běhu
  constexpr bool d = isGreater(-1, v);// chyba, nelze vyhodnotit v kompilačním čase
}
```

Standard C++17 dále také umožňuje tzv. „if constexpr“. Tato vlastnost dovoluje evaluaci podmínky a vyřazení všech ostatních, které nabývají false hodnot. Celá větev v kódu je postupně inicializována pouze a jen v závislosti na dané podmínce. Mechanismus tak nabízí zoptimalizování výkonu, zvýšení rychlosti doby běhu a daleko větší kontrolu chyb zachycených již při překladu. [12]

3.6 Lambda výrazy

Lze definovat jako formulaci vracející funkční objekt. Použity byly poprvé v C++11, přinášejí nový způsob vytvoření anonymního objektu funkce. Typicky se používají k zapouzdření kódu, který je využit jinými algoritmy nebo asynchronními metodami. [20]

Lambda výrazy, zkráceně používané jako lambda, je notace uráčená ke generování funkčního objektu, které jsou často nenáročné ke kopírování. Lambda výraz se skládá z několika částí, jako je klauzule zachycení, seznam volitelných parametrů, volitelných výjimek, návratového typu a těla pro daný výraz. [21]

[=] zachycení lokálních proměnných, parametrů hodnotou

[&] zachycení lokálních proměnných, parametrů referencí

[x, &y] zachycení objektů: x hodnotou, y referencí

[this] this reference

Výrazy se nejčastěji využívají v případech, kdy není vhodné použít funkce. Taková situace může specificky nastat při zachycení proměnných nebo definování metod, které jsou deklarované pouze jako lokální. Funkce totiž jako takové nemohou zachytit již definované lokální proměnné nebo být samy takto definovány, cože je přední pozitivum v používání lambda notace. Na druhou stranu lambdy nelze přetěžovat, v takovém případě jsme nuceni použít funkce. [1] [14]

Z důvodu efektivity a správnosti v kódu se preferuje zachycení daných parametrů pomocí reference v případě, že lambda výraz bude deklarován lokálně. Z hlediska náročnosti bude ve velké většině efektivnější a méně náročné používání reference namísto hodnoty. Naopak v případě nelokálního použití se vyvarujme aplikovat referenci a raději použijme samotnou hodnotu. Důvod je prostý, reference či ukazatele by neměly existovat v programovém cyklu déle než jen ve svém vlastním rozsahu. Zachycení lokální proměnné u lambda výrazu přes její hodnotu, která poté přestane existovat v daném rozsahu, se vyhýbáme nedefinovanému chování či relevantní chybě. [19]

3.6.1 Preferování lambda výrazu před std::bind

Lambda výrazy přinesly také značně úspornější paměťové řešení, přičemž dosahují i daleko rychlejších překladových časů, než tomu bylo s dosavadním používáním šablony bind. Z těchto praktických důvodů by od standardu C++11 neměl být funkční objekt std::bind v efektivním kódu za žádných okolností používán. [17]

Od C++14 lze použít auto typ jako parametr dovolující polymorfické výrazy.

```
auto size = [](const auto &x) { return x.size(); };
```

C++17 dovoluje evaluaci lambda výrazu v době překladu.

```
constexpr auto lambda = [](auto a, auto b)
{
    return std::is_same_v<decltype(a), decltype(b)>;
};
```

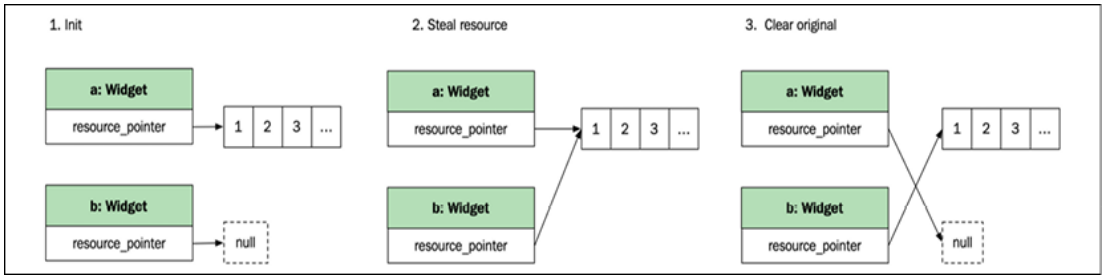
3.7 Efektivní přesun zdrojů

V určitých případech však zachycení hodnoty nebo reference u lambda funkcí není přesně to, co pro konkrétní problém je považováno za vhodné řešení. V dané situaci mohou být například objekty typu unikátního ukazatele, které nelze zkopírovat do jiného. Řešení std::move z C++11 a jeho pozdější rozšíření v následujícím standardu tak lambda funkcím umožnuje přímý přesun objektu a dále ho ve výrazu používat. Přesouvání samotného objektu je také užitečnější, neboť prováděná operace tak zcela eliminuje redundantní kopírování. [22]

```
auto ptr = std::make_unique<int>(x);
auto lambda = [ptr = move(ptr)]()
{
    // použití ptr
};
```

3.7.1 Funkce std::move

Umožňuje přesun celého objektu a zacházení s ním jako s dočasnou r-value hodnotou. Obecný benefit takového případu nastává, pokud nechceme kopírovat potencionálně velké objekty a tím tak pošetřit některé zdroje. Podstatné je si uvědomit, že moved-from objekt by se poté neměl nadále využívat ve smyslu čtení, neboť by v programu vyvolal neočekávané chování, avšak může se do něj přiřadit hodnota nová. Obecné použití je především u objektů, které jsou drahé pro kopírování, ale levné na přesunutí. [12] [23]



Obrázek 3 Přesun zdrojů daného objektu [24]

3.7.2 Perfect forwarding

Nechť deklarujeme funkci, která předává určité argumenty a volá jinou vnořenou funkci. Pokud je zde očekávaný parametr s l-value referencí, ale mylně bude vkládána r-value, program nepůjde přeložit. Pro takový případ bychom museli deklarovat další funkce, které by odpovídaly požadavkům, jež by se vzájemně přetěžovaly. Takovýmto neefektivním způsobem by nebezpečně a neadekvátně začala narůstat délka našeho kódu. Objevovat se také začnou prostory k chybám a příliš časté přetěžování funkcí bude působit komplexně. Tato situace byla vyřešena od vydání C++11 a to s pomocí std::forward, jelikož umožňuje správně zachovávat univerzální přístup k daným parametrům. Použitím std::forward se přetypují parametry funkce dle kategorií l-value nebo r-value v závislosti nad tím, jak jsou předávány. Celý mechanismus tak umožnuje validní a přímé poskytnutí argumentů jedné funkci (r-value jako r-value, l-value jako l-value) a obecně je definován pod pojmem perfect forwarding. [23]

3.8 Další užitečné vlastnosti

3.8.1 Fold Expressions

Přináší od standardu C++17 při deklaraci variadických šablon určité zjednodušení funkčních parametrů s použitím binárních operátorů. Variadické šablony poskytují mechanismus definování proměnlivého počtu argumentů, a jsou tak vhodné v situacích, kdy dopředu není tento počet znám. Patřičné využití je zejména při vývoji C++ knihoven ke zpracování funkcí a zajištění flexibility kódu. [25]

Specializace se dělí do kategorií: přímé a rekurzivní.

- **Přímá** je použita v případech, kdy není třeba provést zpracování jednotlivých argumentů.
- **Rekurzivní** v případě, pokud se zpracovávají jednotlivé argumenty. [26]

Syntax variadické šablony

```
template <typename... Args> class C;
```

Následuje příklad výpisu parametrů s použitím variadických šablon. Specializace je zde přes rekurzivní volání.

C++11/14

```
template <typename T>
void output(const T var)
{
    std::cout << var;
}
template <typename T, typename... Args>
void output(T &&var, Args &&...vars) // forwarding reference
{
    std::cout << var << " ";
    output(std::forward<Args>(vars)...);
}
```

Od **C++17** stejná funkcionalita, ale významné zjednodušení.

```
template <typename... Args>
void output(Args &&...args)
{
    ((std::cout << args << " "), ...);
}
```

3.8.2 Structured binding declaration

Od C++17 nám tato vlastnost umožňuje deklarovat několik proměnných inicializovaných například ze tříd přes member functions, map, polí, párů a dalších. Notace je jednak zpřehledněná, a také dochází k odstranění zbývajících zdrojů neinicializovaných proměnných.

Ve kvalitě daného objektového kódu nebude žádný výkonnostní rozdíl, ve srovnání s použitím explicitního složeného objektu, avšak použitím této struktury dosáhneme výhod v expresivnosti dané programátorské myšlenky. Níže jsou názvy proměnných vázány na první a druhý element mapy. Výhodou je dosažení notační jednoduchosti a snadného použití. [12]

```
std::map<std::string, int> m { {"k1", 1}, {"k2", 2}, {"k3", 3} };
for (const auto &[key, value] : m)
{
    std::cout << key << " " << value << std::endl;
}
```

3.8.3 Inline variables

Poskytují formální způsob, jak definovat globální proměnné v hlavičkových souborech.

Inicializace probíhající mimo strukturu.

C++11/14

```
struct C1
{
    const static std::string s;
};
const std::string C1::s = "String";
```

Před zavedením inline varibles bylo nutné provést inicializaci mimo strukturu. Data ve struktuře C1, jsou konstantní pro celý životní cyklus programu. Překladači bylo nutné dát určité zdroje a čas, aby proměnou inicializoval. [27]

S pomocí inline proměnné je možné samotnou inicializaci provést přímo ve struktuře, přičemž není podstatné, kde přesně má být definována. Tímto bude eliminována předchozí nevýhoda. Inline proměnné jsou však na druhou stranu inicializovány v tzv. thread safe módu.

Pokud se při překladu následujícího kódu (gcc, -std=c++17) podíváme na výslednou binární formu překladu a zaměříme se na strukturu C2, v komentáři zjistíme:

C++17

```
struct C2
{
    inline const static std::string s = "String";
};

cmp BYTE PTR [rip+0x2fe1],0x0   # 404048 <guard variable for S2::s[abi:cxx11]>
```

Nevýhodou je, že překladač musí provést tzv. thread safe check a inicializaci, v případě, že se k dané proměnné bude přistupovat. Samotný překladač totiž nedokáže odhadnout, kdy přesně se proměnná inicializuje v případě více překládaných jednotek.

4 C++20

Od vydání standardu C++14 a 17 byl očekáván důležitější milník, který by do jazyka vnesl užitečné programovací konstrukce. Obdobně jako kdysi při vydání standardu C++11, přinesl tento dosud nejnovější standard dlouho žádané rozšíření. C++20 nabízí set důležitých a očekávaných vlastností, které byly do určité míry již dlouhodobě ze strany vývojářů vyžadovány. Některé z nových vlastností již abstraktně samotný autor jazyka formuloval před dlouhou dobou ve své publikaci [3]. Určité rozšíření nově vybraných vlastností se tak očekávají v následujícím standardu C++23. [1]

Vydání standardu C++20 mělo za cíl splnění určitého požadavku, který by opět přinesl způsob, jak programy zefektivnit. Obdobně tedy jako při vydání standardu C++11, jenž přinášel do jazyka žádané programovací konstrukce a definoval tak nové způsoby při vývoji programů. Nový standard rozšiřuje dosavadní vlastnosti jazyka, které ho opět posunují moderním a úspornějším směrem.

4.1 Přehled nových vlastností

Modules – Definování nezávislých komponent programu. Odstraňují problematické závislosti na preprocesoru, opakovaného importu hlaviček a používání makra. Důležitý milník vzhledem k historii hlavičkových souborů, které jsou již zavedeny déle než padesát let. [28] [29]

Attributes – [[likely]] a [[unlikely]] zlepšující v podmínkových výrazech výkon programu.

Compile-time computation – Zavedení nových vlastností pro evaluaci v době překladu. Klíčové slovo consteval, constinit a nově také constexpr pro virtuální funkce.

Concepts – Vylepšení generického kódu a jeho kontroly v době překladu. Podpora ze strany knihoven pro specifikaci požadovaných parametrů.

Coroutines – Rychlé a flexibilní řešení synchronních a asynchronních úloh. V současném standardu však chybí podpora nativních knihoven.

Three-way comparison – Nový operátor porovnávání a jeho možné přetěžování.

4.2 Testování výkonu

Cílem měřitelného otestování a porovnání standardizace C++17 a C++20 je provedení analýzy a vyhodnocení daného výkonnostního pokroku. Ten zde nese význam rychlejšího operačního času (angl. runtime / execution time) programu, přenesení určitého výpočetního výkonu do překladové doby, zrychlení překladového času nebo jeho úspory v paměťové náročnosti.

Dané testovací programy nebyly koncipovány za účelem vytvoření co možná nejvíce rychlých nebo nenáročných algoritmů. Naopak, v celé řadě situací k testovacím účelům byly vhodné případy vytvořených algoritmů účelně zpomaleny. Tímto krokem bylo zacíleno, aby postupně docházelo ke značenému nárůstu asymptotické složitosti a následná výpočetní náročnost programu byla vysoká i pro relativně nízké vstupní parametry.

S těmito provedenými úpravami tak mohl být veškerý důraz kladen na značné zatížení konstrukcí pro získání nezkreslených výsledků. Výpočty provedené z vytvořených algoritmů mohou snadno běžet i několik minut a tím poskytnout spolehlivě měřitelné a statisticky vyhodnotitelné výsledky. V takovémto způsobu implementace se testovací programy primárně odlišují od standardních nebo běžně používaných programů, které by v optimálních podmínkách běžely pouze v řádech několika stovek milisekund.

Testovány byly nové vlastnosti, jež už z podstaty mají charakter, který vylepšuje v různých směrech danou programovou výkonnost. V novém standardu je totiž také řada nových konstrukcí, které zde nemají vyhrazenou použitelnost za účelem navýšení výkonnostního charakteru. Ty nově v řadě případů poskytují programátorům zlepšení generického kódu, určité zjednodušení a čitelnost nebo dosažení úspornějšího kódu.

Kromě samotného testování je také provedeno u vhodných příkladů určité vysvětlení kódu, a to na úrovni binárních forem, které použité překladače generují. Analýza forem zde vychází ze standardu C++20 za účelem zkoumání rozdílů, které tyto jednotlivé překladače v nových konstrukcích vygenerují a jak se poté v dané formě odlišují.

Programy určené k testování, jež byly překládány ve vybraných překladačích (viz. dále), mají upravenou formu pro příslušný standard, který je jim umožněn. Tím je zde myšleno, že v počátku byl vytvořen program pro standard C++20, který byl následně několika

postupnými kroky převeden do standardu C++17. Nové vlastnosti se zde nevyskytují, jinak by ani z principu překlad nebyl možný.

Všechny zhotovené programy rovněž nedisponují knihovnami třetích stran, podporující jakékoliv navýšení nebo úpravu výkonu daných algoritmů. Tím je nastavena přesná specializace pro zanalyzování čistého výkonu, který je jen a pouze závislý na příslušném standardu jazyka.

Otestování samotného výkonu programů vždy závisí na tom, jaký konkrétní pokrok nám nové vlastnosti poskytují, jelikož obecně vždy přinášejí jen určitá zlepšení do specifické výkonnostní oblasti. V jiných ji poté z tohoto principu nemá smysl testovat, neboť výkon by zůstal nepozměněný.

4.2.1 Způsob měření výsledků

U dané vlastnosti je vždy explicitně vyjádřeno, jaký druh testování byl prováděn. Před zahájením samotného otestování vybraných programů probíhaly přípravné fáze zahrnující velké množství zkušebních experimentů a pokusů. Tímto záměrem je, aby nová vlastnost byla v programu pro daný efekt vhodně použita a mohla být považována za reprezentativní. V případě přezkoumání toho, že by byla použita nedostatečně efektivním způsobem, mělo pak za následek provedení úprav daného programu.

Pro zanalyzování výkonu byly použity tři odlišná testovací zařízení s různými operačními systémy, které jsou i s hardwarovými specifikacemi podrobněji popsány v tabulce dále. V případě podpory použitých nástrojů k překladu je rovněž důležité zmínit, že nikoliv všechny překladače v době testování disponovaly úplnou podporou posledního standardu C++20. Tyto nejznámější a zde použité překladače jsou jmenovitě Clang, g++ a MSVC[7]. U těchto překladačů je velmi podstatné mít alespoň minimální verzi zmiňovanou v konfiguraci, neboť u starších verzí by mohla být podpora nového standardu C++20 buď pouze experimentální nebo zcela postrádající.

4.2.2 Použité techniky v měření

Při testování standardů z hlediska měření operačního času programu byly použity výsledné hodnoty, pro něž byl vytvořen vlastní benchmark. Ten získá statistické vzorky z hlediska naměřeného minima, maxima a aritmetického průměru odečítající tyto dva

[7] Používaný ve vývojovém prostředí Microsoft Visual Studio.

extrémy, přičemž právě tento je použit do následné analýzy dat. Vytvořený benchmark při jednom spuštění programu nabere celkem deset vzorků.

Stanovení operačního času zde poskytuje velmi přesná a testery používaná knihovna Chrono. Ta je navržena tak, aby měřila hodnoty velice precizně a zároveň neutrálně i s ohledem na používání různých systémů využívající odlišné metody.

Testovací bloky pro naměření rychlosti algoritmu jsou zabaleny do funkcí z knihovny přes std::chrono::high_resolution_clock::now(), které zaznamenaly vždy počátek a konec výpočtu konkrétního algoritmu. Získané výsledky pak byly následně vypisovány v konzoli v časových jednotkách nanosekund.

Měření a získávání vzorků bylo pokaždé prováděno v izolovaném prostředí při nezkreslených podmínkách. To zahrnovalo snížení ostatních běžících procesů na pozadí na úplné minimum, zakázání síťového provozu, odmítnutí několika počátečních vzorků při měření a udržování si určitého odstupového času mezi měřeními tak, aby nedocházelo k navýšení výkonu procesoru a jeho přetaktování.

V případě měření překladového procesu (User + System) byl použit v terminálu parametr -ftime-report resp. příkaz time (/usr/bin/time) na Linuxovém prostředí a velikost výstupu preprocesorů přes parametr -E | wc -c. Na operačním systému macOS byl za účelem získání přesných zdrojů použit nástroj GNU Time [30]. V prostředí MS Visual Studiu je pak možné navolit možnosti zobrazování podrobností build reportu přímo v samotném IDE. Obdržené výsledky pak byly vypisovány v časových jednotkách sekund, v případě MSVC zase milisekund. Výstupy preprocesorů jsou poté zaznamenány v bajtech. U referenčních hodnot je zobrazován aritmetický průměr, pokud není řečeno jinak.

V práci bude dále podrobně rozebíráno téma dosažených a statisticky zpracovaných výsledků měření. Z tohoto důvodu zde nebudou explicitně uváděny použité bloky C++ kódu. Všechny tyto zdrojové kódy jsou plně k dispozici v elektronické verzi přílohy závěrečné práce.

4.3 Hardwarové konfigurace

V přehledu níže jsou uvedeny všechny hardwarové konfigurace použité při testování výkonnostního porovnání. Pro přehlednost je u každé z nich nastaveno unikátní *označení* používané u výsledků v kapitolách dále.

Tabulka 2 Konfigurace PC – Windows

Označení	PC – Windows
Operační systém	Windows 10 Home 64bit
Procesor	Intel Core i5-4590CPU, 4CPUs 3.3GHz
Paměť	16 GB RAM, 1600MHz DDR3
Grafická karta	NVIDIA GeForce GTX 960, 2 GB
Pevný disk	SSD 250 GB

Tabulka 3 Konfigurace Notebook – Ubuntu

Označení	Notebook – Ubuntu
Operační systém	Ubuntu 20.04.4
Procesor	AMD RYZEN 7 3700U 4CPUs, 2.3GHz
Paměť	8 GB RAM, DDR3 2400MHz
Grafická karta	AMD Radeon RX Vega 10 Graphics
Pevný disk	SSD 512 GB

Tabulka 4 Konfigurace MacBook – macOS

Označení	MacBook – macOS
Operační systém	macOS Montrey 12.2
Procesor	2,6 GHz Quad-Core Intel Core i7
Paměť	16 GB 2133 MHz DDR3
Grafická karta	Intel HD Graphics 530 1536 MB
Pevný disk	SSD 256 GB

Překladače:

g++ verze 11.2.0

Clang verze 13.0.0 a Clang Apple verze 12.0.5[8]

MSVC verze 19.30 - Microsoft Visual Studio Community 2022

[8] Dostupné pouze u sestavy s označením macOS.

5 Optimalizační atributy

Mezi nově přidanou funkcionalitu do C++20 spadají atributy [[likely]] a [[unlikely]], které dovolují překladači, respektive provedené optimalizaci, upravit generovanou formu tak, aby se při evaluaci podmíněného větvení typu if-else vykonávala značně efektivněji. Programátor tak poskytuje určitou výhodu pro překladač spočívající v tom, která část podmínky je prováděna s vyšší pravděpodobností.

Celá idea použití vyplývá z toho, že podmíněné větvení ve velmi častých případech není rozloženo s rovnoměrnou pravděpodobností. Specifická podmínka totiž z pravidla bývá evaluována s určitou převahou než ostatní. Tato situace se nejen v rozsáhlých projektech vyskytuje nezřídka a potencionální využití této nové funkcionality může vyústit ve značně lepší rychlostní výsledky. Samotné atributy tak ovlivňují pouze rychlost vykonání algoritmu při běhu programu, nikoliv však jeho překladovou dobu.

Atributy je možné využít v případech evaluace větvení, tzn. například v if-else blocích, u příkazu switch a také v ternárních výrazech.

Syntax

```
if (condition) [[likely]]
{
      // kód
}
else (condition) [[unlikely]]
{
      // kód
}
```

5.1 Získání vzorků

Naměřené výsledky, v tomto případě operační čas, vždy ovlivňuje specificky nastavený vstupní parametr v každém programu. Ten poté stanovuje velikost zpracovávaných prvků, případně se jedná o opakování určitých iterací ve smyčce. Každé testování, s použitím vlastního benchmarku, pak proběhlo vždy nejméně pro tři různé vstupní parametry. Konkrétní program byl vždy spouštěn pro vstupní parametry alespoň třicetkrát. To ve výsledné množině výsledků znamená získání a vyhodnocení nejméně tří set různých vzorků pro jeden konkrétní program.

5.1.1 Překlad

Programy využívající možnosti ze standardu C++20 byly přeloženy s parametry -std=c++20 a nastavenou optimalizací -O3. V případě programů přeložených ve standardu C++17 byl použit parametr -std=c++17 používající rovněž stejnou optimalizaci -O3.

5.1.2 Zvolené testovací algoritmy

Pro zvolenou testovací sadu bylo v konečném počtu otestováno osm různých programů ve standardu C++20 (využívající optimalizační atributy) a identický počet pro standard C++17. Programy vždy prováděly určitý výpočet v netriviálním rozsahu, a to s dobou běhu v řádu sekund až po několik minut pro zajištění určité přesnosti získaných výsledků.

Vytvořené algoritmy i za tímto účelem využívají rekurzivní výpočty, jelikož rekurze jako taková zde primárně plní funkci pro využití velice náročných výpočtů a velmi náročného větvení i při hodnotách s poměrně nízkými vstupními parametry. Výsledky algoritmu jsou uloženy do proměnné typu volatile pro zabránění vedlejších efektů.

Přehled použitých algoritmů

Výpočet násobených prvků z vektoru

Evaluace hodnoty cos(x) [31]

Nalezení elementu v daném vektoru

Fibonacciho posloupnost

Určení minimálního prvku v poli

Nejdelší společná sekvence znaků dvou řetězců

Konstrukce binárního stromu

Průchod 2D polem

V následující analýze jsou zobrazovány výsledky překladačů g++ případně Clang. Překladač MSVC není ve výsledcích uveden, neboť získané výsledky všech daných algoritmů vykazovaly mezi C++20 a C++17 jen velmi zanedbatelné hodnoty v intervalu 1 až 1,5 procenta, které se dají spíše považovat za určitou odchylku v měření. Podrobnosti jsou tak následně k tomuto tématu uvedeny v rozboru binárních forem.

Následující tabulka zobrazuje výkonností rozdíl testovacího programu, který vykázal jeden z nejvíce pozoruhodných nárůstů. Algoritmus určuje možný počet pro nejdelší společnou sekvenci znaků ze dvou řetězců, přičemž také inkrementuje svou hodnotu, jestliže je některý z řetězců prázdný.

V prvním sloupci tabulky jsou hodnoty vstupních parametrů, jež následuje statisticky vyhodnocený průměrný operační čas obou verzí algoritmů. Poslední sloupec vpravo zobrazuje procentuální zrychlení C++20 s použitými atributy oproti verzi ve standardu C++17.

Tabulka 5 Nejdelší společná sekvence znaků

Sestava		Notebook – Ubuntu	
Překladač		g++	
Algoritmus: Nejdelší společná sekvence znaků dvou řetězců			
Parametr	Čas v C++20	Čas v C++17	Výkonnostní nárůst
A = 23; B = 14	30.405 s	41.311 s	26.40 %
A = 24; B = 14	42.966 s	64.603 s	33.49 %
A = 25; B = 14	55.529 s	100.782 s	44.90 %
A = 27; B = 14	148.144 s	237.791 s	37.70 %
A = 29; B = 14	376.807 s	512.608 s	26.49 %

V tabulce dále je naopak poukázáno na nejmenší naměřený výkonnostní poměr. S použitím optimalizačních atributů však stále dosahujeme na určitý nárůst. Naměřeny byly taktéž hodnoty algoritmu za podmínek, kdy jsou nastaveny všechny používané atributy do přímé negace [9]oproti optimalizované verzi. Opačné použití tak záměrně a cíleně poukazuje na možný výkonnostní pokles, který by mohl nastat, nakonfiguroval by ho tímto neopatrným způsobem samotný vývojář.

Algoritmus vyhledává element v daném seřazeném vektoru prvků a záměrně se pomaleji posouvá blíže k hledané hodnotě na základě porovnání s ostatními prvky. Parametrem je pak počet N hledaných prvků v pokaždé stejně velkém poli.

[9] Změna [[likely]] na [[unlikely]] a naopak

Naměřené hodnoty byly zpracovány do statistického mediánu.

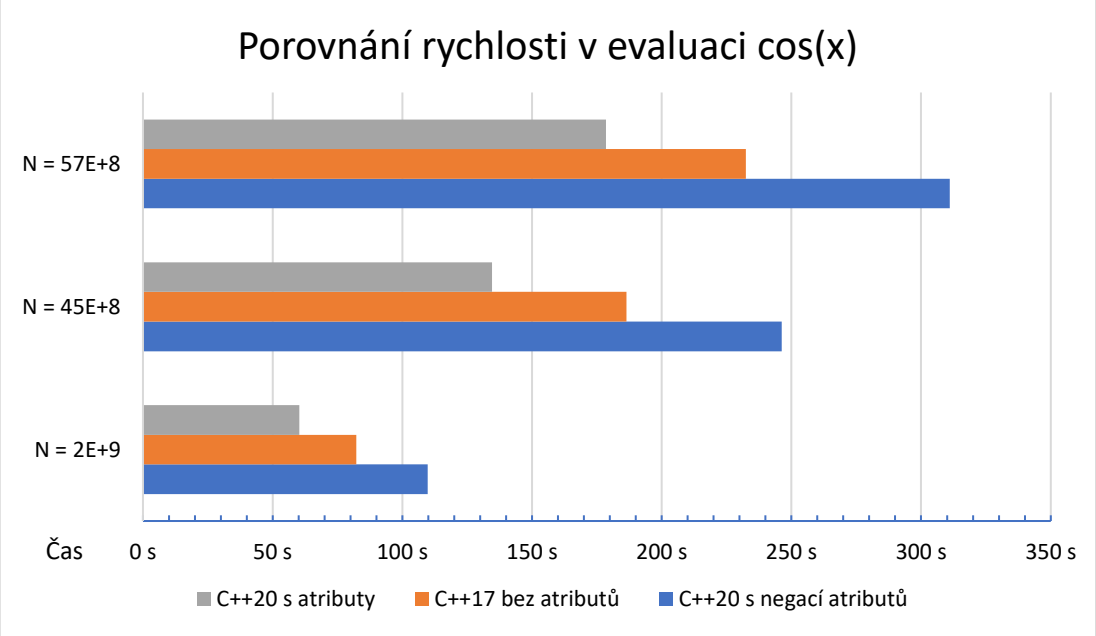
Tabulka 6 Hledání elementu ve vektoru

Sestava		Notebook – Ubuntu	
Překladač		g++	
Algoritmus: Nalezení elementu v daném vektoru			
Parametr	Čas v C++20	Čas v C++17	Čas v C++20 s negací atributů
N = 5	3.782 s	4.060 s	4.924 s
N = 14	10.343 s	11.011 s	13.826 s
N = 30	21.822 s	22.936 s	23.859 s

Pro algoritmus provádějící výpočty hodnot cosinus [31] byl obdobně jako v předchozím případě sledován také výkonnostní poměr atributů v jejich opačném hledisku. Pro standard C++20 jsou optimalizační atributy také použity a otestovány v přímé negaci oproti již zoptimalizované verzi a výsledky byly zaneseny do grafu. Problematika je tak závislá na nastavení od vývojáře, jakým způsobem optimalizaci naplánuje a provede, neboť při takto popsaném nastavení se časová rychlost může zpomalit i do řádu násobku. Parametr poté určuje počet provedených výpočtů, přičemž na vstupní hodnotu cosinu se v průběhu aplikuje drobná dekrementace zlomkem této konstanty.

Sestava Notebook – Ubuntu

Graf 1 Evaluace hodnoty cos(x)



V tabulce níže je přehled naměřených výsledků algoritmu minimálního prvku v poli. Počet elementů je ve všech případech konstantní, parametr zde určuje počet provedených iterací. Pole procházíme celé od shora dolů, přičemž dostaneme-li se k poslednímu prvku, hledání je ukončeno. Ve zbylém případě vracíme minimum z aktuálního pole a prvku vráceného z přechozího rekurzivního volání. Bez použití optimalizačních atributů se velmi zřetelně potřebný operační čas navyšoval.

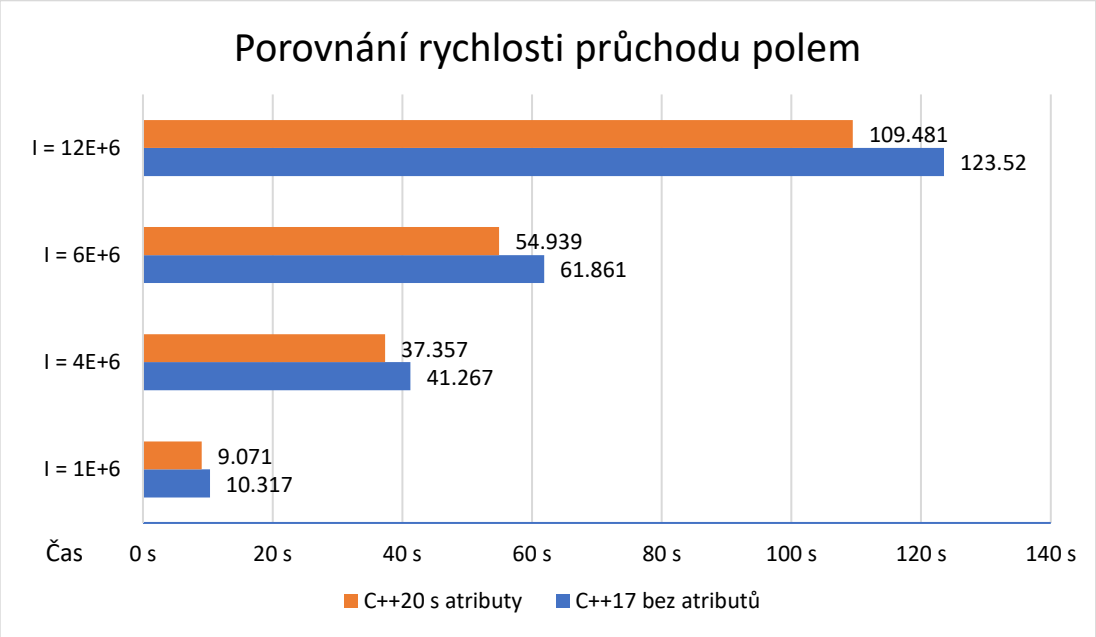
Tabulka 7 Minimální prvek v poli

Sestava		PC – Windows	
Překladač		g++	
Algoritmus: Určení minimálního prvku v poli			
Parametr	Čas v C++20	Čas v C++17	Výkonnostní nárůst
I = 9E+4	11.943 s	16.132 s	25.96 %
I = 2E+5	26.523 s	35.837 s	25.98 %
I = 6E+5	79.646 s	108.377 s	26.50 %

V následujícím grafu jsou zaneseny hodnoty testovacího algoritmu, jenž kompletně prochází čtvercovou maticí s konstantním řádem. Parametr iterace znační celkový počet jejího procestování. Průchod začíná na prvním řádku a sloupci, posouvání probíhá vždy o jednu pozici doprava. Dostaneme-li se na poslední sloupec, provede se daná inkrementace řádku. Pokud by byl počet řádků vyšší, než je řád matice, algoritmus bude tímto ukončen.

Sestava PC – Windows

Graf 2 Průchod 2D polem

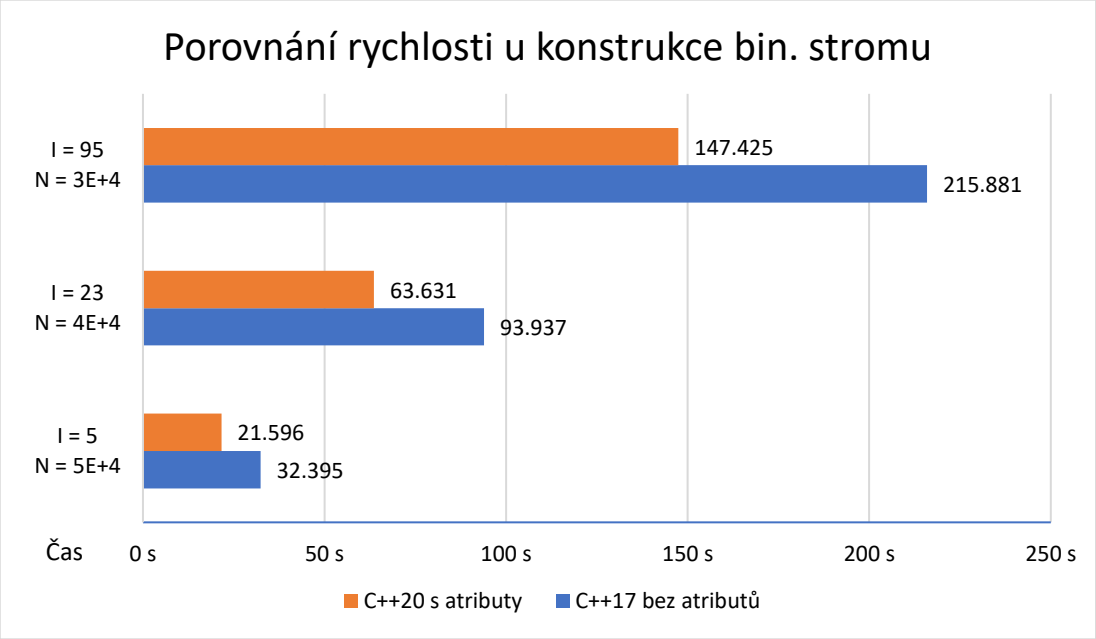


Daný proces má za cíl dvoudimenzionálním polem pouze procházet, nikoliv v něm dále operovat. Jeho obecný princip, zvláště ten v novém standardu, je následně možné použít k určení dimenze vektorového prostoru, transponování matice a dalších operací.

Binární strom je v programování velmi oblíbená datová struktura. Testovaný algoritmus provádí úplné sestavení takového stromu, přičemž celá jeho konstrukce proběhne až do velikosti N v různých iteracích. Potomci rodičů jsou přiřazováni pod ukazatel na základně své velikosti. Výsledky u optimalizačních parametrů jsou opět velmi pozoruhodné.

Sestava PC – Windows

Graf 3 Konstrukce binárního stromu



Algoritmus představený dále reprezentuje velmi náročné výpočty v plovoucí desetinné čárce. Vykonávaný proces cílí na simulaci komplikovaných a zdlouhavých výpočtů dat z dynamické struktury s požitím datových typů, jež mají značnou velikost a přesnost. Vstupní parametr určuje náročnost výpočtů a v průběhu se snižuje, je-li stále kladný.

Tabulka 8 Evaluace prvků z vektoru

Sestava		Notebook – Ubuntu	
Překladač		g++	
Algoritmus: Výpočet násobených prvků z vektoru			
Parametr	Čas v C++20	Čas v C++17	Výkonnostní nárůst
A = 2500.0	8.959 s	13.761 s	34.89 %
A = 4000.0	23.098 s	34.452 s	32.95 %
A = 5500.0	43.544 s	64.199 s	32.17 %

5.1.3 Binární forma překladu

Assembler kód x64 byl vygenerován a analyzován v sestavě Notebook – Ubuntu.

MSVC

Ve fázi testování a měření byl zaznamenán jen velmi zanedbatelný výkonnostní nárůst u MSVC překladače, který výstupy s pomocí atributů nijak neoptimalizoval, tj. ponechával je beze změn. Při pokusech také byla odzkoušena řada optimalizačních přepínačů, které IDE nabízí, případně jejich různé kombinace. U následné analýzy binárních forem, které překladač vygeneroval, bylo zjištěno, že v době testování tyto atributy neměly žádný dopad na výslednou generovanou formu na rozdíl od ostatních použitých překladačů.

G++

U provedené analýzy generovaných forem z assembleru se v kódu zaměřujeme primárně na danou výpočetní funkci, proto byl testovací benchmark či jiný redundantní kód odebrán. U překladače g++ celý binární výstup s použitím atributů ve všech případech narostl a u některých dokonce i na trojnásobek své původní velikosti v závislosti na četnosti použití.

Překladač některé bloky duplikuje, přičemž inkrementuje jejich registry. Bloky jsou také jinak seřazeny v závislosti, jak daný atribut nastavujeme. Obecně provádí více častých MOVL instrukcí a používá jiné typy registrů, než je tomu tak bez daných optimalizačních atributů. Ovlivněná je rovněž také instrukce JE, jež kontroluje stav jednoho nebo více stavových příznaků v registru a provádí skok na cílovou instrukci.

Tyto instrukce jsou nově upraveny tak, že při evaluaci je nyní celý běh v bloku uzpůsoben nastaveným atributům. Následný instrukční skok pro vyhodnocení nejprve proběhne do místa v registru s podmíněným příkazem, a bude se tedy přesně řídit vůči nastavené optimalizaci. Tímto jsou také nově bloky jinak přeuspořádané a následně generují rychlejší kód v přímé závislosti na našem nastavení. Vygenerované binární formy jsou sice v takovýchto případech o něco delší, výhodou je však značně rychlejší podmínková evaluace.

Pokud však atributy nastavíme vůči pomalejšímu výkonu, překladač se striktně řídí naší volbou a vyhodnocuje se přednostně nastavená podmínka, která však s velkou pravděpodobností nebude splněna a pokračovat bude na následující. To má také za následek několikanásobné zvětšení počtu potřebných instrukcí pro vykonání a tím také značné zpomalení.

Toto chování pozorovatelné z binární formy vysvětluje, proč je poté výsledný operační čas o tolik delší, jsou-li optimalizační atributy nastaveny dle takovéhoto případu[10].

Clang

Optimalizátor tohoto překladače používá atributy s ohledem na generovanou binární formu značně úsporněji. Při porovnání obou výsledných forem standardů se u překladu nemění velikost výsledného binárního výstupu. To má také za následek, že binární formy jsou o něco přehlednější a stručnější, než tomu tak bylo u předchozího překladače. V některých případech při použití atributů byl dokonce výsledný binární soubor úspornější.

Vygenerovaný kód funguje na poněkud obdobném principu jako v předchozím případě. U posledního standardu je při použití atributů hlavní změna oproti C++17 na instrukci JL/JGE, která provádí podmíněné skoky do odlišných bloků. Tím vhodně nastavené atributy omezují redundantní skoky a počet vykonaných instrukcí je tak řádově nižší.

Při použití optimalizačních atributů překladač taktéž používá na konkrétních místech odlišné registry. Výhodou použitého překladače je, že nutně negeneruje další bloky registrů, ale dokáže si vygenerovaný kód a řízení jeho skoků lépe zoptimalizovat.

Ukázka části assembler výstupu programu „Fibonacciho posloupnost“ překladače Clang.

C++20 s optimalizací atributů.

```
xorl            %ebp, %ebp
cmpl            $2, %edi
jl              . LBB0_3
movl            %edi, %ebx
```

C++17

```
xorl            %ebp, %ebp
cmpl            $2, %edi
jge             . LBB0_2
movl            %ebx, %ecx
jmp             . LBB0_4
```

Z výstupu je zřejmé, že jsou prováděny odlišně podmíněné skoky do daných bloků.

[10] Viz. výsledky měření výše

V následujícím přehledu je vyjádřen poměr rychlosti algoritmu, jenž vrací dle vstupního parametru n-tý člen pro Fibonacciho posloupnost, a také algoritmu společné sekvence znaků. U obou použitých překladačů byl ve vygenerovaných spustitelných formách zaznamenán patrný výkonnostní nárůst, jsou-li použity ve standardu C++20 nové optimalizační atributy.

Tabulka 9 Porovnání rychlosti Fibonacciho posloupnosti

Sestava		Notebook – Ubuntu		
Algoritmus: Fibonacciho posloupnost				
Překladač	g++		Clang	
Parametr	Čas v C++20	Čas v C++17	Čas v C++20	Čas v C++17
N = 48	13.030 s	14.268 s	13.534 s	15.590 s
N = 50	34.083 s	37.861 s	35.477 s	40.895 s
N = 53	145.459 s	162.435 s	150.221 s	188.194 s

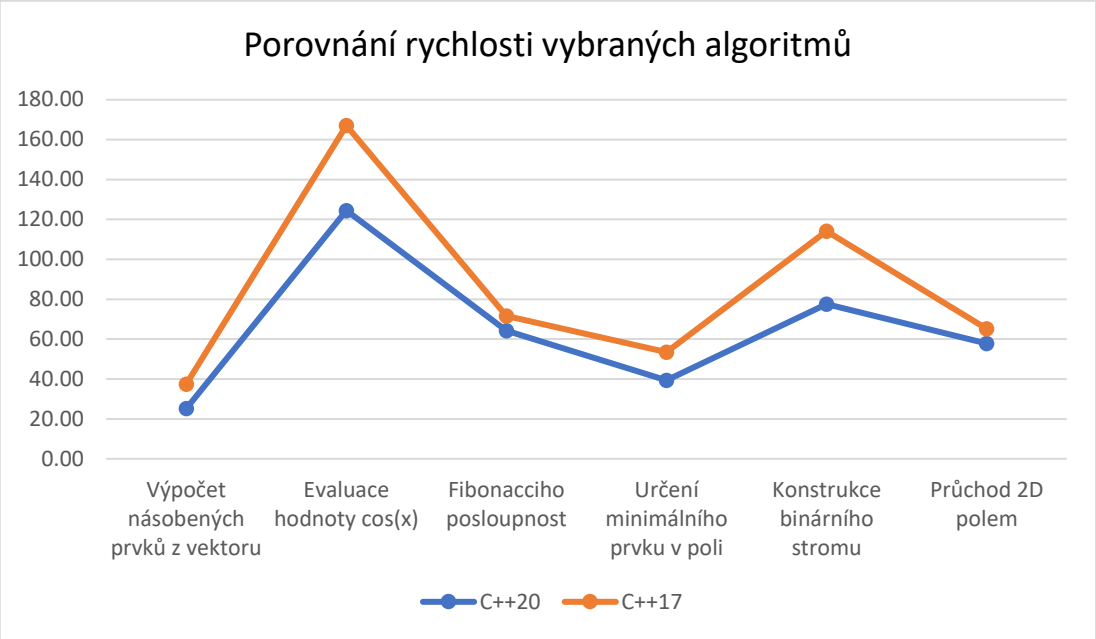
Tabulka 10 Porovnání rychlosti společné sekvence znaků

Sestava		Notebook – Ubuntu		
Algoritmus: Nejdelší společná sekvence znaků dvou řetězců				
Překladač	g++		Clang	
Parametr	Čas v C++20	Čas v C++17	Čas v C++20	Čas v C++17
A = 25; B = 14	55.529 s	100.782 s	107.588 s	119.815
A = 27; B = 14	148.144 s	237.791 s	254.166 s	291.857
A = 29; B = 14	376.807 s	512.608 s	667.486 s	745.678

5.1.4 Souhrn

Graf níže napřímo porovnává výsledné operační časy programů přeloženými v C++20 a C++17 s překladačem g++. Do množiny zobrazovaných výsledků nebyl zahrnut program s nejrychlejším a nejpomalejším nárůstem. Zanesené hodnoty jsou zobrazovány jako průměr ze získaných výsledků. To znamená, že vstupní parametry v rámci dvou verzí daného algoritmu vždy spadají do ekvivalentního intervalu. Levý sloupec pak představuje dobu běhu programu v jednotách sekund.

Graf 4 Přehled výkonnostního porovnání



Z testovaných programů byl zjištěn pozoruhodný nárůst výkonu s použitím optimalizačních atributů oproti dispozicím standardu C++17.

Ve výsledném efektu se jedná o velmi užitečný nástroj, avšak je také důležité jeho obezřetné použití. Při neopatrném nastavení, jak bylo patrné z výsledků, se výkon může velmi rychle zpomalovat. Před samotnou implementací je tak vhodné provést analýzu daného kódu a jeho důkladné otestování. Optimalizační atributy nejsou zcela určeny používat na místech, kdykoliv jen kód provádí evaluaci podmíněných výrazů. Jejich plný potencionál je primárně v úsecích kódu provádějící značnou operační složitost.

6 Přenesení evaluace výpočtů do překladové doby

Nová klíčová slova standardu C++20:

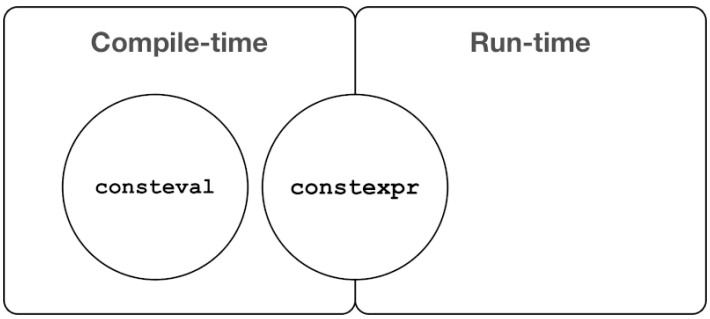
Consteval – tzv. immediate function. Striktně garantuje evaluaci při překladu.

Virtual constexpr – nová možnost evaluace virtuálních funkcí v době překladu. Doposud byl dostupný jen runtime polymorfismus.

Již od standardu C++11, který zavedl klíčové slovo constexpr, existuje možnost evaluace funkcí nebo proměnných přímo v době překladu. Deklarovaná constexpr funkce avšak uživateli tuto vlastnost s jistotou přímo negarantuje. Je pouze schopna být takto využita za určitých, resp. splnitelných podmínek a s konstantními parametry.

Tuto poněkud vágní představu se rozhodli autoři (The ISO C++ committee) jasně vymezit, a tak bylo nově zavedeno klíčové slovo consteval, které přímo garantuje, že daná funkce bude nutně evaluována v době kompilace. Pokud by tento případ nemohl být z určitých důvodů proveditelný, program tak následně nebude možné přeložit. Novou možností opět dostáváme o něco větší kontrolu nad daným kódem a případně zachycení chyb už během překladu.

Za poznámku stojí, že v počátcích testování připravovaných programů byl mimo jiné také ihned experimentálně odzkoušen výkonnostní rozdíl v použití klíčových slov constexpr (C++17) a consteval (C++20). Zde však žádný již z principu totožných úkonů s jistotou nenastává. Klíčová slova se tak v konečném důsledku primárně odlišují v přístupu jejich konkrétního používání.



Obrázek 4 Vztah mezi consteval a constexpr [32]

6.1 Cíle testování

Vzhledem k tomu, že se standardy C++20 a C++17 od sebe nijak neodlišují z časového hlediska rychlosti překládaných algoritmů, využijeme nové rozšířené možnosti z posledního standardu k přenesení výpočtů do překladu a nekonkurenčně je porovnáme s dobou běhu programu v C++17. Tyto jednotky jsou však vzájemně přímo neporovnatelné, což však ani není stanoveným účelem měření.

Cílem je tak danému čtenáři předat ideu a motivovat ho tím, jak určité výpočty přenesené do překladové doby efektivně zrychlí jeho aplikaci, a tím dosáhnout lepších výsledků v operačním čase. Obecně totiž platí, že v každém komplexnějším programu se s jistotou najde prostor, do kterého by se takováto funkcionalita nasadila.

Z pohledu koncového uživatele to znamená, že aplikace po již dokončeném sestavení nemusí při každém požadavku o výsledek opakovaně vyhodnocovat vybrané operace. Představme si možné dopady v globálním měřítku. Analyticky předem vhodně zvolené výpočty přenesené do překladové doby mohou aplikaci zrychlit, aniž by bylo nutné provádět velice náročné zásahy do kódu. Značnou výhodou je například po dokončeném otestování správnosti relativně snadné nasazení těchto vlastností do projektu. Pokud by nám však některá možnost z určitých důvodů nevyhovovala, lze původní kód relativně snadno zpátky přetransformovat.

6.1.1 Měření a počet vzorků

U testovacích programů tak budou využita pouze nová klíčová slova C++20 a hlavním předmětem zájmu je zde doba překladu. Naproti tomu u standardu C++17, který těmito novými možnosti nijak nedisponuje, samotná evaluace výsledků bude probíhat až v samotném běhu programu, jenž je sledovanou hodnotou.

Výsledky jsou poté vyhodnoceny tak, že výpočet algoritmu v C++20 následně vždy poběží již konstantní rychlostí s úvahou delšího času překladu. V porovnání pak u C++17 uvidíme, o jaký časový rozdíl by aplikace v C++20 byla při samotném běhu rychlejší.

Opět je důležité zmínit, že tyto jednotky nejsou vzájemně přímo porovnatelné a stanoveným cílem je tak využít zcela nový způsob zrychlení aplikace, který ještě doposud nebyl přímo takto proveditelný.

V případě, že se jedná o testování zaměřené na výsledný čas překladu, byl daný program přeložen právě třicetkrát pro jeden vstupní parametr s nastaveným přepínačem -fconstexpr odpovídající složitosti operace. Pro měření operačního času v C++17 byl

použit zmiňovaný vlastní benchmark se třemi různými vstupními parametry. Každý program byl spuštěn pro jeden vstupní parametr právě třicetkrát. Ve výsledné množině tak bylo získáno devět set různých vzorků pro konkrétní program.

6.1.2 Testované algoritmy

S klíčovým slovem consteval

Fibonacciho posloupnost

Evaluace determinantu matice

Minimální, maximální a složená čísla v poli

Počet řešení pro N-Queens problém

Rychlé řazení prvků

S klíčovým slovem virtual constexpr

Hodnota binomického koeficientu

Test prvočíselnosti

Variace řazení polí

Pokud překladač vyhodnotí výsledek funkce při kompilaci, poté je již v samotném běhu hodnota známa a nezabere žádný výpočetní výkon navíc. Rovněž se také zachytí případné chyby už během této fáze, čímž se dopředu vyhýbáme potencionálně riskantním důsledkům. Tyto argumenty jsou jedny z předních důvodů, proč takovéto funkce realizovat.

Následující přehled vyhodnocuje dobu překladu za použití klíčového slova consteval oproti evaluaci hodnot při běhu programu vykazující v obou případech jejich průměrný čas k výpočtu. Poslední sloupec napravo představuje, o kolik bychom aplikaci zrychlili, převádíme-li výpočet ve standardu C++20 do překladu.

Průměrný operační čas a překladová doba potřebná pro získání výsledku n-tého členu Fibonacciho posloupnosti.

Tabulka 11 Fibonacciho posloupnost

Sestava		Notebook – Ubuntu
Překladač		g++
Algoritmus: Consteval – Fibonacciho posloupnost		
Parametr	Překlad v C++20	Běh v C++17
N = 44	2.97 min	4.593 s
N = 45	4.84 min	7.367 s
N = 46	8.12 min	11.753 s

Pro tabulky níže se v případě C++20 během kompilace inicializuje jednorozměrné, resp. v případě determinantu dvourozměrné statické pole, které bylo pro algoritmus přímo navrženo. Deklarované consteval funkce poté provedou požadovaný výpočet dle vstupního parametru. Ten zde představuje velikost N daného pole, resp. počet provedených iterací. V případě C++17 je pak popisovaný průběh vytvořen při běhu programu.

Tabulka 12 Určení prvků v poli

Sestava		MacBook – macOS
Překladač		Apple Clang
Algoritmus: Consteval – Minimální, maximální a složená čísla v poli		
Parametr	Překlad v C++20	Běh v C++17
N = 4E+4	7.83 min	0.262 s
N = 5E+4	12.19 min	0.398 s
N = 6E+4	17.05 min	0.565 s

Tabulka 13 Determinant matice

Sestava		Notebook – Ubuntu
Překladač		g++
Algoritmus: Consteval – Evaluace determinantu matice		
Parametr	Překlad v C++20	Běh v C++17
I = 2	6.44 min	0.296 s
I = 3	9.36 min	0.455 s
I = 4	13.04 min	0.594 s

Následný komplexní návrh systému je tak potřeba vhodně uzpůsobit a volit rozložení daného výkonu tak, aby výsledný poměr nebyl k jedné straně zaměřený přespříliš či neadekvátně náročný.

Výsledky níže kromě samotného porovnání potencionální úspory poukazují především na rozdíl v rychlosti evaluovaných výsledků danými překladači, které jsou mezi sebou znatelně odlišné. Důležité je, že všechny provedené překlady v této kapitole nebyly ovlivněny optimalizačními přepínači, které by tyto časové hodnoty dále výrazně ovlivnily. Parametry zde v obou případech představují vstupní velikost polí.

Tabulka 14 N-Queens problém

Sestava			PC – Windows			
Algoritmus: Consteval – Počet řešení pro N-Queens problém						
Překladač	g++		Clang		MSVC	
Parametr	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17
N = 10	0.23 min	0.023 s	1.07 min	0.026 s	0.79 min	0.024 s
N = 11	1.24 min	0.127 s	5.92 min	0.140 s	4.70 min	0.127 s
N = 12	7.47 min	0.711 s	35.69 min	0.827 s	28.35 min	0.728 s

Tabulka 15 Rychlé řazení prvků

Sestava			PC – Windows			
Algoritmus: Consteval – Rychlé řazení prvků						
Překladač	g++		Clang		MSVC	
Parametr	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17
N = 500	1.63 min	0.132 s	9.55 min	0.112 s	8.03 min	0.142 s
N = 600	2.87 min	0.226 s	17.26 min	0.192 s	15.14 min	0.240 s
N = 700	4.62 min	0.356 s	28.85 min	0.302 s	24.75 min	0.375 s

V dále uvedeném přehledu z testovaných programů používají třídy evaluaci jejich virtuálních funkcí v době překladu. Deklarované funkce poté v rámci programu vykonávají obdobnou funkcionalitu, odlišují se však ve způsobu dané implementace. Parametry v případě první tabulky představují vlastní výpočet kombinačního čísla $\binom{n}{k}$. Pro zbylé dva pak představuje parametr rozsah čísel, respektive velikost pole.

Tabulka 16 Binomický koeficient

Sestava			PC – Windows			
Algoritmus: Virtual constexpr – Hodnota binomického koeficientu						
Překladač	g++		Clang		MSVC	
Parametr	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17
N = 29; K = 11	2.02 min	0.187 s	11.12 min	0.193 s	8.09 min	0.352 s
N = 30; K = 11	3.16 min	0.295 s	17.36 min	0.305 s	13.76 min	0.556 s
N = 31; K = 11	4.93 min	0.459 s	27.12 min	0.475 s	22.04 min	0.864 s

Tabulka 17 Test prvočíselnosti

Sestava			PC – Windows			
Algoritmus: Virtual constexpr – Test prvočíselnosti						
Překladač	g++		Clang		MSVC	
Parametr	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17
N = 20000	1.96 min	0.490 s	10.75 min	0.531 s	10.09 min	0.544 s
N = 27500	3.79 min	0.919 s	20.27 min	0.999 s	19.02 min	1.028 s
N = 35000	5.96 min	1.470 s	32.49 min	1.600 s	30.48 min	1.631 s

Tabulka 18 Řazení polí

Sestava			PC – Windows			
Algoritmus: Virtual constexpr – Variace řazení polí						
Překladač	g++		Clang		MSVC	
Parametr	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17	Překlad v C++20	Běh v C++17
N = 2000	2.24 min	0.088 s	7.27 min	0.085 s	5.58 min	0.121 s
N = 3000	5.52 min	0.197 s	16.46 min	0.188 s	14.06 min	0.270 s
N = 4000	9.56 min	0.351 s	29.89 min	0.334 s	24.35 min	0.477 s

6.1.3 Adekvátní zrychlení

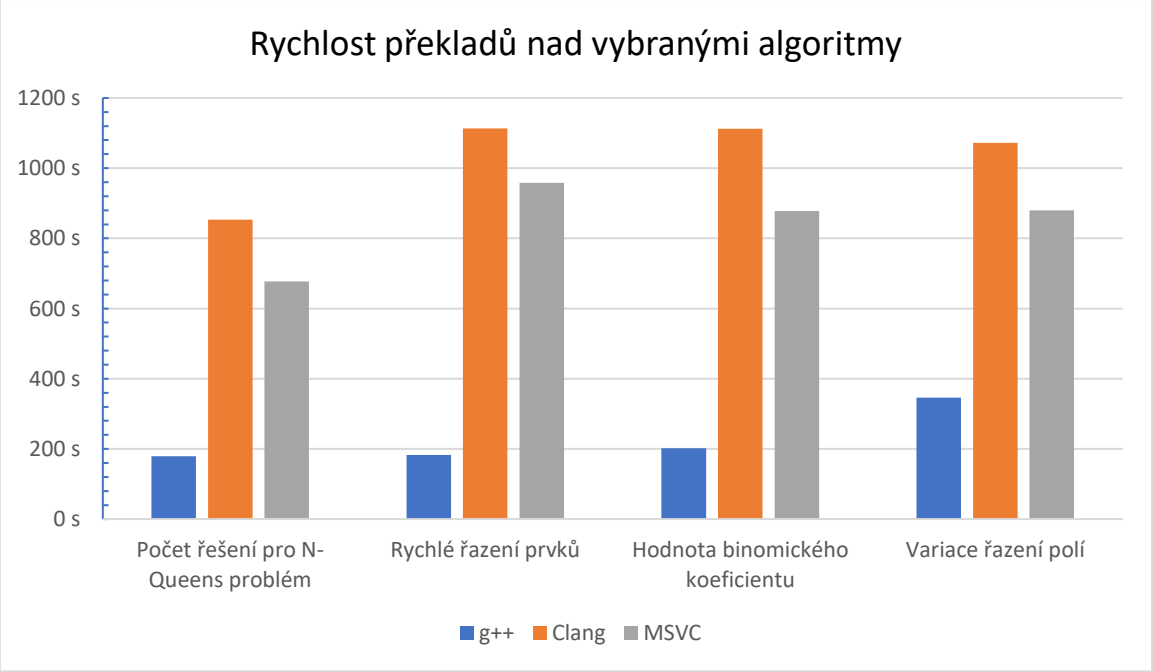
Ze všech obdržených výsledků je patrné zvážit určitou reflexi při optimalizaci. Překlad by neměl být neadekvátně dlouhý vůči následnému zrychlení. V programu by tak měl být nastaven určitý kompromis, kdy za cenu přijatelných nákladů překladu výpočty efektivně zrychlíme.

Tato získaná zlepšení jsou obecně i v tomto malém měřítku pro aplikaci již citelně znatelná. Pokud bychom však celou situaci převedli na daleko rozsáhlejší projekty či provozní aplikace a upravily vhodné výpočty s těmito novými možnostmi, kód by se stal o něco více výkonným.

V následujícím grafu jsou srovnávány použité překladače z hlediska rychlosti překladu používající daná klíčová slova z nového standardu. Naměřené hodnoty vstupních parametrů byly následně zprůměrovány a zaneseny do grafu. Vstupní parametry zde odpovídají hodnotám uvedené v předešlých tabulkách.

Sestava PC – Windows

Graf 5 Analýza rychlosti překladů



Z grafu je velmi patrné, že překladač g++ dosahoval znatelně nejrychlejších překladů a jeho optimalizace na nový standard je zde bezprecedentní. Jako druhý v pomyslném pořadí následuje překladač MSVC. Za opravdu citelně nejdelšími časy

překladače Clang mohla být také neúplná podpora z vývojářské strany, neboť v době otestování, nebyla zcela optimální viz. [33]

6.1.4 Virtual contexpr a consteval v assembleru x64

Dané argumenty consteval funkce je vždy nutné deklarovat jako konstanty. Pokud bychom se tímto pravidlem striktně neřídili, funkce nepůjde přeložit a překladač zobrazí chybu s tím, že daný parametr není deklarován jako konstantní výraz.

Narozdíl u constexpr funkce tato podmínka zmíněna výše nezpůsobí chybu při překladu. Funkce by však byla s úplnou jistotou evaluována až v době běhu programu. Takovéto chování je jednotné a je vyžadováno všemi použitými překladači.

Pokud je však funkce napsána se záměrem evaluace v době překladu, tak při prozkoumání výsledné binární formy se daná funkce ve výstupu nijak nevyskytuje. To znamená, že do dané proměnné probíhá pouze už jen MOVL instrukce s již vypočteným výsledkem. Takovéto chování bylo zaznamenáno rovnocenně u constexpr i consteval funkce.

Virtual constexpr

Jednotlivé překladače za následujících podmínek přistupují k deklaracím u constexpr a virtual constexpr funkcí s trochu odlišnými rozdíly. Při zkoumání assembleru u překladače g++ a vyzkoušení možných kombinací jednotlivých nastavení zůstával binární výstup prakticky totožný. U parametru, který je však jakkoliv znám dopředu, se provede rovnou MOVL instrukce a výpočet proběhne v době překladu.

```
auto a = virtual_constepxr_funkce(); // compile-time
```

U překladače Clang je situaci trochu odlišná. Pokud danou proměnnou přímo nedeklarujeme u constexpr virtuální funkce jako konstantu, bude nadále přítomna ve výstupu binární formy a její výsledek bude znám až při samotném běhu.

```
auto a = virtual_constepxr_funkce();       // run-time
const auto b = virtual_constepxr_funkce(); // compile-time
```

Zde může nastat otázka, co kdybychom potřebovali dále proměnou používat jinak než jen v režimu konstanty, aniž bychom ji museli poté přetypovávat nebo ošetřovat takovéto chování pokaždé a na všech místech. Zde nastává další rozdíl s použitím consteval, jenž dovoluje, aby deklarovaná proměnná nebyla nutně konstantní. Tím je snazší dále pro používání, a přesto si stále zachová evaluaci při překladu. Dosavadní nevýhodu však je, že virtuální funkce nemohou být typu consteval.

U MSVC je situace obdobná jako v přechozím případě, avšak překladač je v tomto směru ještě o něco striktnější a požaduje, aby daná proměnná byla nutně typu constepxr. Při prozkoumání binárního výstupu tak jedinou možností byla pro přímé provedení MOVL instrukce po překladu právě tato možnost.

```
const auto a = virtual_constepxr_funkce();     // run-time
constexpr auto b = virtual_constepxr_funkce(); // compile-time
```

Tyto různé vlastnosti jsou důležité při následné realizaci, aby exekuce kódu byla provedena tak jak je zamýšleno. Consteval nám však nově dává v tomto ohledu o něco přesnější kontrolu nad kódem.

6.2 Constinit

Klíčové slovo constinit ze standardu C++20 se používá pro staticky deklarované proměnné a za pomoci tohoto klíčového slova je garantováno, že daná proměnná bude inicializovaná v době překladu. Deklarace globálních proměnných, které nejsou vytvořeny v době překladu, pak mohou vést k seriózním chybám, které je později velmi obtížné detekovat. [34]

Používáním nového klíčového slova při deklaraci se takovýchto chyb vyvarujeme. Výhodou namísto inicializace constexpr proměnné je, že hodnotu uloženou v paměti můžeme v průběhu programu kdykoliv měnit.

V binární formě je poté drobný rozdíl, pokud přistupujeme k daným proměnným.

Přímý přesun hodnoty.

```
constexpr int a = 111;
movl    $111, -8(%rbp)
```

Přesun hodnoty přes proměnou.

```
constinit int b = 14;
movl b, %eax
movl %eax, -12(%rbp)
```

7 C++ Modules

Standard C++20 nově přichází s dlouho očekávaným a zcela nově moderním a úspornějším řešením pro knihovny a ostatní jednotky překladu. Moduly tak obecně poskytují zcela nový způsob, jak pracovat s více propojenými jednotkami překladu. Odstraňují rovněž opakovaně vyskytující se problémy, které stále mají běžné hlavičky.

U nich je faktický problém v tom, že mohou vyvolat spoustu neočekávaného chování, jež při importu hlaviček ovlivňovalo pořadí, v jakém byly uspořádány. Tedy například hlavička1 mohla nezáměrně ovlivnit deklaraci a makra patřící pro hlavičku2 a naopak. [35]

Rozveďme příklad problému makra a jeho pořadí. Jakou hodnotu bude mít následující konstanta při importu obou hlaviček? Deklarace jedné entity ve dvou překladových jednotkách může způsobit nekonzistenci a překladač určité chyby nemusí ani nutně zachytit. [36]

```
// Header1.h
#define CONSTANT 0b11000

// Header2.h
#define CONSTANT 159
```

Moduly velmi zásadně odstraňují tyto problémy a obecně omezují nežádoucí chování. Eliminují také problémy makra a definovaného pořadí souboru. Žádný definovaný modul tak nepřeteče do jiného ani nijak negativně neovlivní ostatní v rámci celého procesu překladu. [36]

Dalším již dlouhodobě velmi problematickým a diskutovaným tématem jazyka je doba celého překladu. Pro představu uvažme, pokud bychom jeden konkrétní hlavičkový soubor zahrnovaly do 100 překladových jednotek, kód takového souboru by musel být preprocesorem zpracován stokrát. Tím je tak nově v úplném kontrastu významově stejně funkční modul, které je však zpracováván pouze jednou.

Představíme-li si takovýto problém v globálnějším měřítku, například u složitějších aplikací a systémů, dosahujeme na neúměrně dlouhé časy celého překladu. Tím se značně komplikuje celý vývoj, nehledě pak na rostoucí finanční náročnost celého projektu.

Nejzásadnější přínos spočívá v tom, že jakmile je modul přeložen, uchová se do binární podoby. Takovýto modul je mnohem rychleji zpracováván než hlavičkový soubor, neboť překladač ho jen znovu použije na každém místě, kdekoliv se vyskytuje. Moduly přinášejí daleko lepší vytváření logických struktur, a také nám umožnují import a export jen určitých částí, které daný modul obsahuje. Soubor několika modulů pospolu se poté dá snadno exportovat jako jeden logický balíček. [12] [29]

Pro představu, dalším jednoduchým a demonstrativním příkladem může být klasický helloworld program, který zahrnuje pro výpis knihovnu iostream. Samotný program lze jednoduše napsat na 4-6 řádků, avšak včetně zmiňované knihovny bude mít výstup preprocesor dohromady více než 32000 řádků[11]. Nehledě tedy na rostoucí náročnost, pokud knihovnu importujeme v dalších jednotkách překladu. Používání hlavičkových souborů už je tak v dnešním moderním pohledu poněkud zastaralejším, daleko pomalejším přístupem, jak logicky dělit programové části s náchylnostmi k nežádoucím defektům. [36]

Definování modulu začíná klíčovým spojením: export module „název“. Jeho importování následně definujeme obdobným způsobem: import „název“. Pro část modulu, jenž je určený k exportování například funkce, jmenného prostoru, třídy apod. přidáváme klíčové slovo export. Modul také může být snadno rozdělen do několika logických oddílů, kde poté oddělujeme interface od implementace. [37]

Syntax

```
module název: část1;
export class C{};
export void function(){};
export namespace {}
module název: část2;
import název: část1;
```

[11] Testováno u překladače g++.

7.1 Testování modulů oproti hlavičkovým souborům

Porovnání z hlediska výkonnostních rozdílů nastává u zhotovených programů, které využívají kompletního sestavení buď jen s pomocí modulů (C++20), nebo na straně druhé pomocí hlavičkových souborů (C++17).

U všech řešení testujeme výslednou dobu překladu, ale zaměříme se také na velikost v bajtech ve fázi preprocesoru a na spotřebovanou paměť během procesu překládání (umožňují nástroje na systémech Ubuntu a macOS).

7.1.1 Charakteristika testovaných souborů

Daná řešení ve všech těchto případech nejsou cílená na provádění užitečných výpočtů či algoritmů. Pro testování by nenesly tížený význam. Jednotlivé programy tak využívají import modulů/hlaviček v různém rozsahu. Vytvořené soubory lze charakterizovat jako různé matematické moduly a početní operace, entity užívající hierarchickou dědičnost a jeden obsahově totožný modul, který je pro tížený nárůst využíván několikrát. Deklarován je také export konkrétních tříd, jmenných prostorů nebo vybraných funkcí. Pro oddíly je také u některých vhodných případů rozdělen interface od implementací. Programy dále využívají rozmanitý import používaných knihoven. V konečném důsledku se tedy jedná primárně o testovací programy odlišného rozsahu, zamýšlené k těmto zmiňovaným experimentálním účelům.

K testovacím účelům bylo vytvořeno celkem pět různorodých programů.

Programy využívající moduly byly přeloženy a testovány ve standardu C++20.

- Module1 … Module5

Na straně druhé (algoritmicky totožné) programy využívající hlavičkové soubory vychází ze standardu C++17.

- Header1 … Header5

Všechny soubory jsou z hlediska významové funkčnosti pro všechny tři překladače obdobné. Došlo však k několika drobným úpravám především v části použitých knihoven, kterými dané překladače disponují. V případě testování výsledné doby překladu či spotřebované paměti je pro konkrétní řešení zaznamenáno 30 různých vzorků. Velikost souboru během preprocesing fáze je následně vždy konstantní.

7.1.2 Přehled analýzy testování

Clang

V tabulce jsou uvedeny analyzované hodnoty časů překladu. Rychlostní výsledky v případě používání modulů namísto hlavičkových souborů jsou významově rychlejší.

Tabulka 19 Čas kompilace – Clang

Sestava		MacBook – macOS	
Překladač		Apple Clang	
Program	Čas překladu	Program	Čas překladu
Module1	0.611 s	Header1	2.048 s
Module2	0.935 s	Header2	3.816 s
Module3	0.801 s	Header3	3.837 s
Module4	0.884 s	Header4	3.958 s
Module5	2.810 s	Header5	7.466 s

Následující přehled zobrazuje poměr pro výstupy preprocesorů a průměrnou paměťovou spotřebu během procesu překladu. Zobrazený velikostní poměr mezi jednotlivými programy je s použitím modulů značně úspornější, neboť u hlavičkových souborů preprocesor vkládá obsah opakovaně, kdykoliv se v kódu vyskytuje direktiva „include“. Obdržené hodnoty jsou zobrazeny dle převodu soustavy SI.

Tabulka 20 Velikost výstupu preprocesoru – Clang

Sestava		MacBook – macOS	
Překladač		Apple Clang	
Program	Velikost	Program	Velikost
Module1	2.283 KB	Header1	3.715 MB
Module2	4.890 KB	Header2	7.886 MB
Module3	4.677 KB	Header3	7.215 MB
Module4	3.886 KB	Header4	6.996 MB
Module5	14.300 KB	Header5	13.334 MB

Tabulka 21 Spotřeba paměti procesu – Clang

Sestava		MacBook – macOS	
Překladač		Apple Clang	
Program	Spotřeba paměti	Program	Spotřeba paměti
Module1	65.657 MB	Header1	76.009 MB
Module2	61.850 MB	Header2	72.667 MB
Module3	59.510 MB	Header3	75.361 MB
Module4	68.187 MB	Header4	76.873 MB
Module5	78.117 MB	Header5	92.856 MB

MSVC

Pro získané výsledky byly dále soubory testovány v prostředí MS Visual Studio. Tabulka zobrazuje průměrnou dobu k překladu a zrychlení, kterého jsme dosáhli při využití nových modulů.

Tabulka 22 Čas kompilace – MSVC

Sestava		PC – Windows		
Překladač		MSVC		
Program	Čas překladu	Program	Čas překladu	Procentuální zrychlení
Module1	270.90 ms	Header1	827.56 ms	67.26 %
Module2	364.56 ms	Header2	905.90 ms	59.75 %
Module3	313.60 ms	Header3	858.43 ms	63.46 %
Module4	415.36 ms	Header4	892.00 ms	53.43 %
Module5	361.70 ms	Header5	1047.16 ms	65.45 %

G++

Tabulka níže porovnává programy s hlavičkovými soubory a moduly, jež byly testovány na operačním systému Ubuntu s překladačem g++ z hlediska velikosti výstupů daných preprocesorů.

Tabulka 23 Velikost výstupu preprocesoru – g++

Sestava		Notebook – Ubuntu	
Překladač		g++	
Soubor	Velikost	Soubor	Velikost
Module1	2.382 KB	Header1	2.268 MB
Module2	7.659 KB	Header2	5.574 MB
Module3	6.101 KB	Header3	4.090 MB
Module4	3.996 KB	Header4	8.470 MB
Module5	15.923 KB	Header5	9.607 MB

U programu s označením Header4 byla použita knihovna <bits/stdc++.h>, která zajišťuje různé algoritmické funkce. U modulů ji však pro naše potřeby zcela dostatečně nahradil import modulu iostream, což je také jeden z důvodů, proč jsou tyto dva výsledky v následujících dvou tabulkách mezi sebou tak rozdílné.

Tabulka 24 Čas kompilace – g++

Sestava		Notebook – Ubuntu	
Překladač		g++	
Soubor	Čas překladu	Soubor	Čas překladu
Module1	0.520 s	Header1	1.374 s
Module2	1.020 s	Header2	1.992 s
Module3	0.692 s	Header3	2.008 s
Module4	0.859 s	Header4	5.967 s
Module5	3.658 s	Header5	7.370 s

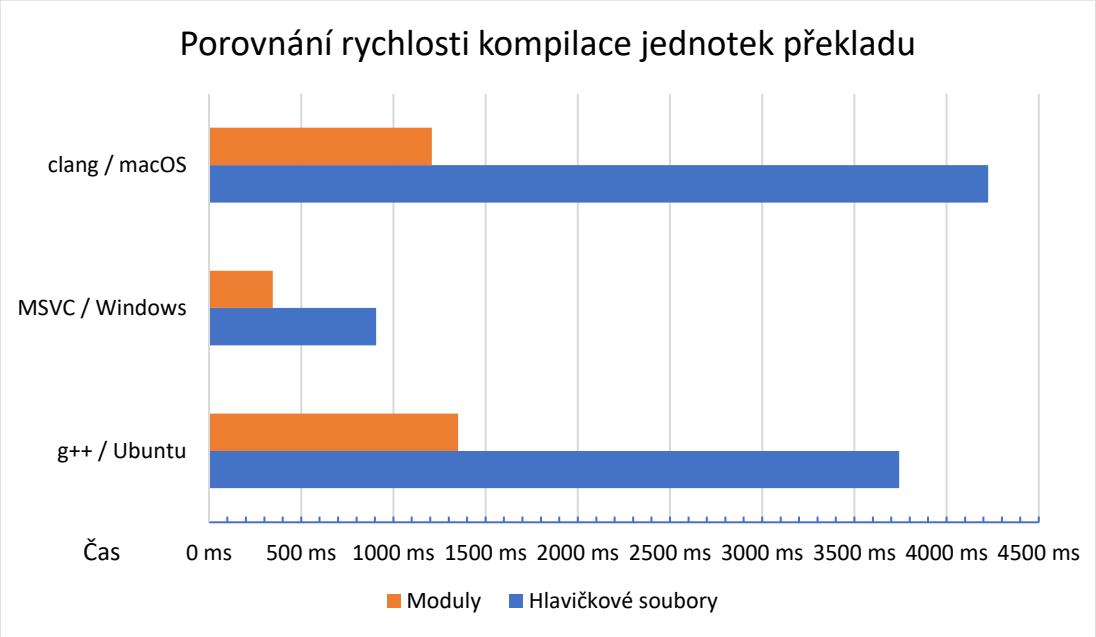
Následující tabulka zobrazuje průměrné hodnoty paměti v megabajtech spotřebované během procesu provedených překladů.

Tabulka 25 Spotřeba paměti procesu – g++

Sestava		Notebook – Ubuntu	
Překladač		g++	
Soubor	Spotřeba paměti	Soubor	Spotřeba paměti
Module1	63.658 MB	Header1	84.534 MB
Module2	81.773 MB	Header2	98.410 MB
Module3	54.630 MB	Header3	84.482 MB
Module4	101.837 MB	Header4	272.368 MB
Module5	124.788 MB	Header5	133.144 MB

7.1.3 Moderní řešení v podobě modulů

Graf 6 Analýza překladů hlaviček a modulů



Graf výše sumarizuje přímé porovnání překladových časů u modulů a hlavičkových souborů napříč všemi otestovanými programy. Z přehledu vyplývá, že u každého použitého překladače došlo k impozantním zlepšením. Moduly tak nově představují rychlé a moderní řešení, které opravuje nedostatky hlavičkových souborů. Jejich postupné zapojování do současných řešení tak velmi značně ovlivní jejich výkon a potřebnou rychlost sestavení, která bývá u provozních aplikacích čím dál více nákladná.

8 Další vybrané vlastnosti C++20

8.1 Coroutines – C++20

Coroutines jsou ve standardu novým modelem, který zavádí kooperativní multitasking, obdobně jako tuto funkcionalitu doposud plnila vlákna. Výhodou nad používání vláken je, že máme mnohem přesnější kontrolu nad celým procesem. Coroutines poskytují zcela nový syntax pro manipulaci v procesu, a na rozdíl od vláken jsou díky takovému přístupu, zejména u rozsáhlých projektů, paměťově více úspornější. [24]

Definice coroutine je de facto určitá funkce, která se v definovaných momentech dokáže pozastavit. Z hlediska coroutines je to vlastně určitá abstrakce, která nám dovoluje tzv. „lazy eveluated code“ a asynchronní programování čistějším způsobem, než tomu tak bylo doposud. Coroutines patřící do C++20 nepoužívají závislosti volání na zásobníku jako běžné funkce. Pozastavují volání tím, že se vracejí zpět k volanému objektu a data, která jsou vyžádána pro pokračování jsou uložena odděleně od zásobníku, a to specificky na haldě. [38]

8.1.1 Designové cíle

V C++20 coroutines nejsou a ani nebyly designovány za účelem toho, abychom je používali na běžných místech, které doposud byly kontrolovány funkcemi s nekomplikovaným způsobem použití v kódu jako doposud. Účelem tedy není poskytnout nástroj, který bude nahrazovat kód v běžném používání tak, aby byl rychlejší. Je to spíše unikátní nástroj pro neobyčejné programové výzvy a moderní algoritmické problémy. Pokud bychom je bezmyšlenkovitě nasadily do aktuálních programů, pravděpodobně bychom si tím spíše jen uškodili. [24]

Funkcionalita pro jazyk je stále ještě relativně nová a vývojáři předních překladačů stále implementují aktualizace a stabilnější vylepšení. U některých překladačů zatím dokonce vlastnost úplně chybí nebo je omezena jen pro experimentální režim. Stále aktuální problém je rovněž ve velmi malé podpoře nativních knihoven. Ty jsou spíše k dispozici od vývojářů třetích stran, což ale nespadá pod oficiální komisi jazyka ani do naší oblasti. Pro zajímavost se jedná například o CppCoro knihovnu, pro asynchronní přístup pak Boost.Asio (core) viz. [39].

Na rozdíl od vláken jsou z hlediska paměťové náročnosti úspornější a díky tomu je možné vytvořit a udržovat více coroutines najednou než udržovat adekvátně stejné množství vláken.

8.1.2 Vlastnosti

Efektivní přepínání kontextu – pozastavení a obnovení coroutine je na stejné náročnostní úrovni jako je například zavolání běžné funkce.

Flexibilita a komplexnost možností – coroutines mají mnoho možností, jak je přizpůsobovat, což dává vývojářům velkou svobodu variability. Rozhodnutí, jak vlastně mají fungovat, drží v rukou vývojáři a určité či nové možnosti se budou jistě teprve v čase objevovat.

Nevyžadují zpracovávat výjimky – mohou být aplikovány i za podmínek, u nichž nejsou použity. Coroutines se definují jako nízko úrovňová funkcionalita, která nemá své využití v obecném měřítku, ale ve specifických odvětvích jako mohou být real-time systémy. [24]

Coroutines po jazykové stránce pro jejich kontrolu přináší následující klíčová slova:

- co_await – operátor suspenduje aktuální coroutine
- co_yield – vrací hodnotu volajícímu a suspenduje coroutine
- co_return – dokončí a uzavře danou coroutine a nepovinně vrací hodnotu [24]

Příklady funkcí pro práci

std::coroutine_handle třída odkazující na stav coroutine, umožnuje její pozastavení, obnovení

std::suspend_never a std::suspend_always očekávaný typ, který se nikdy resp. pokaždé pozastaví

std::coroutine_traits používá se pro tzv. promise

8.1.3 Důležité restrikce u coroutines

- Nelze použít variadické argumenty
- Nelze je deklarovat s pomocí auto typu
- Nelze je evaluovat v překladové době, tj. nemůžeme použít consteval/constexpr
- Nelze aplikovat na main funkci
- Konstruktor a destruktor nemůže být coroutine [24]

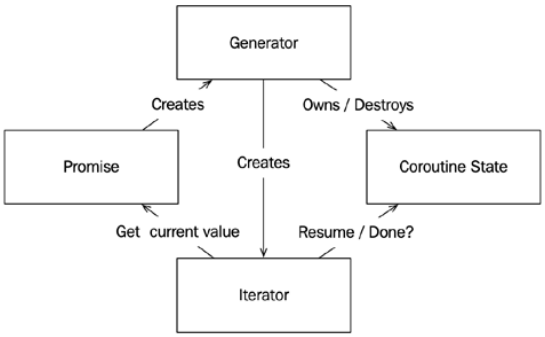
Uvažme následující problém. Pokud definujeme funkci, které vrací vektor o určité velikosti například 100 prvků, a poté chceme vytisknout například jen prvních 10, musíme alokovat celou tuto kapacitu, i když ji nevyužijeme celou. Takovýto obecný problém jednoduše coroutines rozklíčují. Funkce popsaná níže takovýto problém řeší a na základě požadavku poskytne požádaná data, což znamená, že se vytvoří pouze tolik prvků, kolik jich aktuálně požadujeme.

```
Generator<int>
getNums(int start = 0, int step = 1)
{
    auto val = start;
    for (auto i = 0;; i++)
    {
        co_yield val;
        val += step;
    }
}
```

Třída Generator generuje hodnotu při každém zavolání. Vrací objekt pro coroutines se sekvenčním přístupem.

Promise – jde vlastně o controller pro coroutine.

Iterator – interface mezi klientem a třídou Promise. [40]



Obrázek 5 Vztah mezi třídami a stavem coroutine [24]

Ukázka asynchronního přístupu reprezentuje třída task.

```
auto async_function() -> task<int>
{
    // výpočet
    co_return;
}
```

Samozřejmě tímto coroutines nekončí, jejich využití je velmi široké a popsat jejich fungování detailněji by vydalo na samostatnou publikaci.

8.2 Concepts

8.2.1 Definice

Koncepty jsou pojmenovaný set určitých požadavků používaný v generické části kódu. Požadavky mohou být syntaktické, sémantické nebo komplexní. [24]

Návrhové cíle k vytvoření konceptů byly takové, že šablony mohou být použity tak, že v určitých případech je velmi obtížné odhadnout její specifikaci. Daný interface může být příliš generický a tím může nastat mylná představa o tom, co přesně daná funkce vykonává. V takovém případě pak zbývá buď vyhledávat informace v dokumentaci, nebo složitěji v kódu přímo danou implementaci. [24]

Doposud bylo možné omezit šablonu tak, aby akceptovala jen objekty určitého typu s pomocí funkce std::enable_if. Její použití avšak není úplně uživatelsky přívětivé, nehledě na to, že pokud se do šablony dosadil neakceptovatelný typ, tak překladový výpis byl velmi obtížně čitelný. Typové chyby jsou zachyceny v době překladu a běžné překladače podávají v takovýchto případech zprávy, které jsou i pro zkušené vývojáře těžko interpretované. Zjistit tak, kde konkrétně chyba vznikla, bylo neadekvátně náročné, nehledě pak s používáním polymorfismu, který by se stával u velkých projektů náročnější na jeho čitelnost. [41]

Syntax:

```
template <parametry>
concept název = constrains;
```

Definování konceptu v C++20 je nově velmi přímočaré a flexibilní dle konkrétních situací.

```
template <class T>
concept Number = std::is_integral_v<T> || std::is_floating_point_v<T>;
void foo(Number auto t); // použití

template <class T>
concept Small = sizeof(T) <= sizeof(int);
```

8.3 Constrains

Používají se spolu s koncepty definováním klíčového slova requires. Výhodou je opět, že vykonání kódu probíhá v době překladu, a tím se dá snadno předejít nedostatkům, které by jinak vznikly při běhu programu. To nám opět dovoluje vytvářet software méně náchylný k chybám. Tělo klíčového slova constains pak obsahuje logické výrazy, které je možné řetězit konjunkcí, disjunkcí, případně zůstanou atomické. [42]

Syntax:

```
requires { requirements }
requires(parameter list) { requirements }
```

Příklad použití, který při použití šablon hlídá deklaraci metod v dané třídě. [42]

```
concept Person = requires(T p)
{
    typename T::id;
    p.viewStats();
    p.changeInfo();
};
```

Další použití může být nadefinování různých operátorů, které mohou být proměnnou vykonány, volání určitých funkcí, přetypování nebo invokace konstruktoru.

8.4 Three-way comparison

Nový operátor <=> pro porovnání dvou objektů. Dané porovnání má za výsledek tři možnosti, které definují vztah mezi objekty. V případě, že je negativní, je první objekt menší než druhý. V pozitivním případě je situace opačná a pokud je hodnota nulová, jsou si objekty zcela rovny. [42]

Pro fundamentální typy jako je integer, float a podobně to není protazím nijak zvlášť převratná funkce, avšak pro objekty typu string přináší objektivnější výhody. V kódu se použitím operátoru vyhýbáme zbytečnému složitému větvení a tím ho i zpřehledňujeme.

Příklad

```
auto a = „string a“;
auto b = „string b“;
auto res = a <=> b;
```

Při prozkoumání několika obdobných případů na úrovni assembleru x64 u MSVC používá nový operátor instrukci MOVZX, která vyhodnocuje s pomocí strong::ordering daný výraz. Což je tak určitý rozdíl od dosud běžných porovnávacích operátorů, které tuto instrukci neprovádějí.

```
movzx eax, BYTE PTR static std::strong_ordering const
```

V C++17 bychom pro porovnání v definované třídě, která používá například dvě souřadnice, museli definovat alespoň šest přetížení operátorů. Nově tak namísto poněkud refundačního kódu stačí přímo definovat přes nový operátor jediné přetížení, které zastoupí všechny případy použití.

```
auto operator<=>(const Class &) const = default;
```

8.5 Range-based cyklus s inicializací

Vlastnost dovoluje inicializaci dat napřímo v cyklu a jejich následné použití. Výhoda je, že nemusíme data získávat jinde v kódu, ale napřímo a přehledně s nimi pracovat hned v cyklu.

```
for (auto x = getData(); const auto &a : x)
{
    // použití proměnné
}
```

8.6 Knihovna

Nová knihovna span pro standard C++20 elegantně řeší jeden z fundamentálních problémů při práci s indexací, kdy se chybou dostaneme za vymezenou hranici vyrovnávací paměti. Tomuto problému se také jinak říká přetečení bufferu a je to také jeden z hlavních bezpečnostních problémů. Překladač tuto chybu nezachytí a u velkých projektů následné debuggování může být velice časově nákladné. Span na úrovni abstrakce je vlastně ukazatel s velikostí, přičemž tento nápad prezentoval již v devadesátých letech Dennis Ritchie. [1]

U argumentů následující funkce tak nově není potřeba mít proměnnou pro uložení velikosti ukazatele.

```
template <class T>
void foo(std::span<T> p) // span zachovává ukazatel a velikost
{
    for (const auto &x : p)
        std::cout << x << " ";
}

int *p = a; // ukazatel na pole
foo<int>({p, *(&a + 1) - a});
```

Závěr

Tato práce se zabývá z dnešního pohledu moderními a efektivními způsoby programování, které nám současný jazyk C++, jakožto programátorům a vývojářům, nabízí. Pojednává také o široké rozšířenosti jazyka v současném žebříčku programovacích jazyků a dotýká se problematiky z hlediska efektivity náročných projektů.

Z počátku bylo zformulováno krátké historické uvedení, zabývající se primárně vývojem a evolucí, kterou si daný jazyk procházel. Vysvětlena byla důležitost standardizace jazyka, jeho hlavní důvody vzniku a vlastnosti, které se v průběhu let na základě uživatelských požadavků měnily.

V následující části byly představeny ze standardu C++11/14/17 velmi důležité a v současném programování nezbytné konstrukce a vlastnosti, které jsou zde primárně zařazeny vůči dané efektivitě, celkové správnosti a užitečnosti. Daného uživatele tak uvádí do uceleného přehledu, jak se vhodným použitím vyvaruje potencionálním chybám a proč je daná konstrukce v současnosti nezbytná pro celkovou účinnost a správnost.

Pro stanovené cíle práce v testovací části z hlediska efektivity, aktuálně nového standardu C++20, byly zvoleny vhodné konstrukce a vlastnosti, které vykazovaly charakteristiku potencionálního přínosu rychlostních změn. Tyto nově zařazené vlastnosti do posledního standardu byly výkonnostně zanalyzovány na vytvořených programech a pečlivě testovány na třech nezávislých výpočetních sestavách oproti disponujícím možnostem standardu předchozího. Testování probíhalo za použití v současnosti stále nejběžněji používaných překladačů, kde byl také porovnán v rámci testované vlastnosti jejich vzájemný výkonnostní poměr. V důsledku tak testování přineslo přínosné informace o dosaženém výkonu, závislém také na volbě konkrétního překladače.

Práce je však stále určitým náhledem do moderního používání jazyka C++ a jeho celkového efektivního využití, které se stále nově objevují. Obsažený text práce také posloužil jako výchozí zdroj informací a výsledků pro společně vytvořený vědecký článek pojednávající k tématu energeticky efektivních aplikací v data centrech.

Seznam použité literatury

[1] STROUSTRUP, Bjarne. Thriving in a Crowded and Changing World: C++ 2006–2020. Stroustrup.com [online]. 2020 [cit. 2021-11-15]. Dostupné z: https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf

[2] Energy Efficiency across Programming Languages. Greenlab.di.uminho.pt [online]. [cit. 2022-08-19]. Dostupné z: https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf

[3] STROUSTRUP, Bjarne. *The Design and Evolution of C*. Reading: Addison-Wesley Publishing Company, 1994. ISBN 02-015-4330-3.

[4] STROUSTRUP, Bjarne. *A tour of C*. Upper Saddle River, NJ: Addison-Wesley, [2014]. C in-depth series (Addison-Wesley). ISBN 978-0321958310.

[5] ROUPEC, Jan. Objektově orientované programování v C++. Physics.ujep.cz [online]. Univerzita J. E. Purkyně [cit. 2021-11-15]. Dostupné z: http://physics.ujep.cz/~mmaly/vyuka/ruzne/Programovani-D/Cpp03.pdf

[6] STROUSTRUP, Bjarne. *The C++ programming language*. 4th ed. Upper Saddle River: Addison-Wesley, 2013. ISBN 978-0321958327.

[7] VIRIUS, Miroslav. *Jazyky C a C++: kompletní průvodce*. 2., aktualiz. vyd. Praha: Grada, 2011. Knihovna programátora (Grada). ISBN 978-80-247-3917-5.

[8] PRATA, Stephen. *Mistrovství v C++*. Praha: Computer Press, 2001. Všechny cesty k informacím. ISBN 80-7226-339-0.

[9] BENEŠ, Nikola. PB161 Programování v jazyce C++ Přednáška 1. Is.muni.cz [online]. Masarykova univerzita 18.9.2018 [cit. 2021-12-04]. Dostupné z: https://is.muni.cz/el/fi/jaro2020/PB161/um/slidy_2018/s01.pdf

[10] Welcome back to C++ – Modern C++. Microsoft C++, C, and Assembler documentation [online]. [cit. 2021-12-04]. Dostupné z: https://docs.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-170

[11] JOSUTTIS, Nicolai. *C++ standard library: tutorial and reference*. Boston: Addison-Wesley, 1999. ISBN 0201379260.

[12] STROUSTRUP, Bjarne. *A tour of C++. Second edition*. Boston;: Addison-Wesley, [2018]. C++ In-Depth series. ISBN 978-0-13-499783-4.

[13] History of C++. En.cppreference.com [online]. [cit. 2021-12-04]. Dostupné z: https://en.cppreference.com/w/cpp/language/history

[14] CALANDRA, Anthony. Modern C++ features. Github.com [online]. [cit. 2021-12-04]. Dostupné z: https://github.com/AnthonyCalandra/modern-cpp-features

[15] C++11 Overview. Isocpp.org [online]. [cit. 2021-12-04]. Dostupné z https://isocpp.org/wiki/faq/cpp11

[16] Auto (C++). Microsoft C++, C, and Assembler documentation [online]. [cit. 2021-12-04]. Dostupné z: https://learn.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-170

[17] MEYERS, Scott. *Effective modern C++*. Sebastopol, CA: O´Reilly, [2015]. ISBN 978-1-491-90399-5.

[18] Constexpr (C++). Microsoft C++, C, and Assembler documentation [online]. [cit. 2021-12-14]. Dostupné z: https://docs.microsoft.com/en-us/cpp/cpp/constexpr-cpp?view=msvc-170

[19] STROUSTRUP, Bjarne. C++ Core Guidelines. Github.com [online]. [cit. 2021-12-14]. Dostupné z: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

[20] GELLER, Barbara. Back to Basics Lambda Expressions. Github.com [online]. CppCon 2020 [cit. 2021-12-14]. Dostupné z: https://github.com/CppCon/CppCon2020/blob/main/Presentations/back_to_basics_lambda_expressions/back_to_basics_lambda_expressions__barbara_geller__ansel_sermersheim__cppcon_2020.pdf

[21] Lambda expressions in C++. Microsoft C++, C, and Assembler documentation [online]. [cit. 2021-12-14]. Dostupné z: https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170

[22] How to: Create and use unique_ptr instances. Microsoft C++, C, and Assembler documentation [online]. [cit. 2021-12-14]. Dostupné z: https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-170

[23] HAMELIN, Adrien. Quick Q: What's the difference between std::move and std::forward? Isocpp.org [online]. 27.2.2018 [cit. 2021-12-18]. Dostupné z: https://isocpp.org/blog/2018/02/quick-q-whats-the-difference-between-stdmove-and-stdforward

[24] ANDRIST, Bjorn a Viktor SEHR. C++ High Performance: Master the art of optimizing the functioning of your C++ code. 2nd Edition. Birmingham: Packt Publishing, 2020. ISBN 978-1839216541.

[25] Ellipsis and Variadic Templates. Microsoft C++, C, and Assembler documentation [online]. [cit. 2022-01-30]. Dostupné z: https://docs.microsoft.com/en-us/cpp/cpp/ellipses-and-variadic-templates?view=msvc-170

[26] VÍTEK, Stanislav. Variadické šablony, paralelní programování. Cw.fel.cvut.cz [online]. České vysoké učení technické v Praze [cit. 2022-01-30]. Dostupné z: https://cw.fel.cvut.cz/b212/_media/courses/b2b99ppc/ppc-lec-06.pdf

[27] KRÜGLER, Daniel. Inline Variables for the Standard Library. Open-std.org [online]. 27.2.2017 [cit. 2022-01-30]. Dostupné z: https://open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0607r0.html

[28] STROUSTRUP, Bjarne. A minimal solution to the concepts syntax problems. Open-std.org [online]. 6.5.2018 [cit. 2022-03-06]. Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1079r0.pdf

[29] Overview of modules in C++. Microsoft C++, C, and Assembler documentation [online]. [cit. 2022-03-06]. Dostupné z: https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170

[30] GNU Time. Gnu.org [online]. [cit. 2022-09-29]. Dostupné z: https://www.gnu.org/software/time/

[31] C++ attribute: likely, unlikely. En.cppreference.com [online]. [cit. 2022-03-16]. Dostupné z: https://en.cppreference.com/w/cpp/language/attributes/likely

[32] FERTIG, Andreas. C++20: A neat trick with consteval. Andreasfertig.blog [online]. 6.7.2021 [cit. 2022-03-22]. Dostupné z: https://andreasfertig.blog/2021/07/cpp20-a-neat-trick-with-consteval

[33] Compiler support for C++20. En.cppreference.com [online]. [cit. 2022-03-04]. Dostupné z: https://en.cppreference.com/w/cpp/language/attributes/likely

[34] What's the "static initialization order 'fiasco' (problem)"? Isocpp.org [online]. [cit. 2022-03-22]. Dostupné z: https://isocpp.org/wiki/faq/ctors#static-init-order

[35] LISCHNER, Ray. Exploring C++20: The Programmer's Introduction to C++. 3rd Edition. New York: Apress, 2020. ISBN 978-1484259603.

[36] C++20: The Advantages of Modules. Modernescpp.com [online]. 10.5.2020 [cit. 2022-03-22]. Dostupné z: https://www.modernescpp.com/index.php/cpp20-modules

[37] STROUSTRUP, Bjarne. Syntax alternatives for modules. Open-std.org [online]. 20.11.2018 [cit. 2022-04-14]. Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1373r0.pdf

[38] GRIMM, Rainer. From Functions to Coroutines. Github.com [online]. CppCon 2020 [cit. 2022-04-14]. Dostupné z: https://github.com/CppCon/CppCon2020/blob/main/Presentations/from_functions_to_coroutines/from_functions_to_coroutines__rainer_grimm__cppcon_2020.pdf

[39] Overview. Boost.org [online]. [cit. 2022-04-14]. Dostupné z: https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio/overview.html

[40] BAKER, Lewis. C++ Coroutines: Understanding the promise type. lewissbaker.github.io [online]. 5.9.2018 [cit. 2022-04-14]. Dostupné z: https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type

[41] TURNER, Jason. C++ Best Practices: 45ish Simple Rules with Specific Action Items for Better C++. Victoria: Leanpub, 2020. ISBN 979-8822105607.

[42] HORTON, Ivor a Peter VAN WEERT. Beginning C++20: From Novice to Professional. 6th Edition. New York: Apress, 2020. ISBN 978-1484258835.

Seznam obrázků

Seznam tabulek

Přílohy

- Zdrojové kódy otestovaných algoritmů (komprimovaný soubor).
- Statisticky vyhodnocené výsledky v této práci níže.

Tabulka 5.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Nejdelší společná sekvence znaků dvou řetězců“, překladač g++/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 23; B = 14	29.354	30.989	30.405	30.180	30.794	30.477	0.499	1.732
100	A = 24; B = 14	41.915	43.986	42.966	42.014	43.743	43.219	0.797	1.957
100	A = 25; B = 14	53.456	56.649	55.529	54.573	56.462	56.026	1.077	2.046
100	A = 27; B = 14	143.616	151.891	148.144	146.875	150.174	148.077	2.237	1.591
100	A = 29; B = 14	366.298	384.574	376.807	371.038	381.917	379.336	6.151	1.720

Tabulka 5.B: Algoritmus „Nejdelší společná sekvence znaků dvou řetězců“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 23; B = 14	39.872	41.983	41.311	40.964	41.759	41.510	0.629	1.606
100	A = 24; B = 14	62.448	65.948	64.603	63.635	65.764	64.818	1.222	1.994
100	A = 25; B = 14	98.520	102.325	100.782	99.449	102.042	101.094	1.329	1.390
100	A = 27; B = 14	232.015	255.919	237.791	232.959	238.724	236.23	6.504	2.883
100	A = 29; B = 14	487.489	526.603	512.608	501.583	520.048	517.039	11.348	2.333

Tabulka 6.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Nalezení elementu v daném vektoru“, překladač g++/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 5	3.635	3.988	3.788	3.679	3.896	3.782	0.112	3.123
100	N = 14	10.059	10.488	10.322	10.224	10.462	10.343	0.136	1.392
100	N = 30	21.638	22.159	21.864	21.699	22.028	21.822	0.167	0.805

Tabulka 6.B: Algoritmus „Nalezení elementu v daném vektoru“, překladač g++/c++20 (s negací atributů), sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 5	4.788	5.139	4.947	4.866	5.036	4.924	0.105	2.251
100	N = 14	13.364	14.442	13.888	13.711	14.149	13.826	0.31	2.354
100	N = 30	22.335	24.401	23.578	22.690	24.145	23.859	0.722	3.230

Tabulka 6.C: Algoritmus „Nalezení elementu v daném vektoru“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 5	3.86	4.223	4.062	3.959	4.181	4.060	0.123	3.192
100	N = 14	10.154	12.901	10.961	10.191	11.324	11.011	0.819	7.88
100	N = 30	21.769	24.595	22.967	22.115	23.554	22.936	0.834	3.831

Tabulka Graf.1.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Evaluace hodnoty cos(x)“, překladač g++/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 2E+9	59.454	62.077	60.315	59.807	60.580	60.015	0.873	1.526
100	N = 45E+8	133.288	136.100	134.535	133.377	135.778	134.067	1.095	0.858
100	N = 57E+8	174.691	184.951	178.511	176.652	180.423	177.650	2.890	1.706

Tabulka Graf.1.B: Algoritmus „Evaluace hodnoty cos(x)“, překladač g++/c++20 (s negací atributů), sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 2E+9	105.088	114.791	109.805	107.516	113.251	108.536	3.085	2.961
100	N = 45E+8	235.481	262.786	246.255	238.768	251.754	247.549	7.974	3.413
100	N = 57E+8	251.757	362.147	311.100	261.711	358.860	314.210	40.192	13.618

Tabulka Graf.1.C: Algoritmus „Evaluace hodnoty cos(x)“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 2E+9	79.700	84.178	82.232	80.290	83.762	82.823	1.666	2.135
100	N = 45E+8	181.97	190.147	186.461	184.737	189.073	186.373	2.423	1.370
100	N = 57E+8	228.302	234.597	232.434	231.457	234.269	232.544	1.935	0.877

Tabulka 7.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Určení minimálního prvku v poli“, překladač g++/c++20, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 9E+4	11.900	12.213	11.943	11.904	11.934	11.907	0.090	0.802
100	I = 2E+5	26.453	26.804	26.523	26.470	26.518	26.504	0.097	0.385
100	I = 6E+5	79.398	80.003	79.646	79.473	79.852	79.578	0.200	0.265

Tabulka 7.B: Algoritmus „Určení minimálního prvku v poli“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 9E+4	16.002	16.797	16.132	16.009	16.122	16.035	0.237	1.552
100	I = 2E+5	35.551	36.575	35.837	35.573	36.303	35.610	0.381	1.121
100	I = 6E+5	106.667	109.667	108.377	106.836	109.032	108.981	1.081	1.052

Tabulka Graf.2.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Průchod 2D polem“, překladač g++/c++20, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 1E+6	8.993	9.243	9.071	8.994	9.168	8.998	0.095	1.105
100	I = 4E+6	36.447	38.434	37.357	36.521	38.145	37.368	0.789	2.228
100	I = 6E+6	54.030	55.460	54.939	54.142	55.300	55.248	0.559	1.074
100	I = 12E+6	108.039	110.188	109.481	108.168	110.085	110.020	0.894	0.860

Tabulka Graf.2.B: Algoritmus „Průchod 2D polem“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 1E+6	10.177	10.399	10.317	10.206	10.379	10.350	0.082	0.846
100	I = 4E+6	41.209	41.311	41.267	41.251	41.294	41.267	0.028	0.073
100	I = 6E+6	61.092	63.363	61.861	61.135	62.177	62.100	0.694	1.184
100	I = 12E+6	122.106	124.565	123.520	122.283	124.306	123.778	0.901	0.769

Tabulka Graf.3.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Konstrukce binárního stromu“, překladač g++/c++20, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 5; N = 5E+4	21.512	21.686	21.596	21.552	21.635	21.598	0.051	0.249
100	I = 23; N = 4E+4	63.493	63.772	63.631	63.523	63.701	63.665	0.089	0.148
100	I = 95; N = 3E+4	147.088	147.681	147.425	147.273	147.632	147.373	0.192	0.137

Tabulka Graf.3.B: Algoritmus „Konstrukce binárního stromu“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	I = 5; N = 5E+4	32.165	33.410	32.395	32.203	32.377	32.306	0.346	1.127
100	I = 23; N = 4E+4	93.835	94.071	93.937	93.852	94.047	93.907	0.087	0.098
100	I = 95; N = 3E+4	214.852	216.476	215.881	215.67	216.246	215.987	0.456	0.222

Tabulka 8.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Výpočet násobených prvků z vektoru“, překladač g++/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 2500.0	8.549	9.700	8.959	8.749	9.022	8.956	0.293	3.456
100	A = 4000.0	22.452	23.829	23.098	22.668	23.648	22.850	0.493	2.253
100	A = 5500.0	42.996	44.917	43.544	43.244	43.742	43.394	0.507	1.229

Tabulka 8.B: Algoritmus „Výpočet násobených prvků z vektoru“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 2500.0	13.238	14.709	13.761	13.316	14.364	13.365	0.555	4.258
100	A = 4000.0	32.738	35.652	34.452	33.089	35.430	34.875	1.076	3.293
100	A = 5500.0	62.195	66.930	64.199	63.006	66.126	63.547	1.571	2.579

Tabulka 9.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Fibonacciho posloupnost“, překladač g++/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 48	12.698	14.028	13.030	12.743	13.150	12.970	0.376	3.048
100	N = 50	33.696	34.411	34.083	33.962	34.275	34.082	0.200	0.619
100	N = 53	144.765	147.246	145.459	144.797	145.828	145.213	0.762	0.552

Tabulka 9.B: Algoritmus „Fibonacciho posloupnost“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 48	13.984	14.587	14.268	14.107	14.425	14.258	0.188	1.392
100	N = 50	37.407	38.254	37.861	37.654	38.113	37.816	0.255	0.711
100	N = 53	161.092	165.835	162.435	161.149	163.752	161.911	1.595	1.035

Tabulka 9.C: Algoritmus „Fibonacciho posloupnost“, překladač clang/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 48	13.515	13.637	13.534	13.519	13.532	13.521	0.034	0.271
100	N = 50	35.318	35.786	35.477	35.395	35.549	35.438	0.123	0.367
100	N = 53	146.211	151.365	150.221	150.109	151.237	150.397	1.417	0.994

Tabulka 9.D: Algoritmus „Fibonacciho posloupnost“, překladač clang/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	N = 48	15.578	15.615	15.590	15.583	15.594	15.587	0.009	0.066
100	N = 50	40.787	41.074	40.895	40.823	40.981	40.859	0.094	0.244
100	N = 53	186.912	189.859	188.194	186.955	188.752	188.355	0.962	0.539

Tabulka 10.A: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Nejdelší společná sekvence znaků dvou řetězců“, překladač clang/c++20, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 25; B = 14	106.173	110.968	107.588	106.459	108.461	107.185	1.384	1.356
100	A = 27; B = 14	246.381	267.563	254.166	247.427	259.37	251.83	7.274	3.016
100	A = 29; B = 14	654.515	674.424	667.486	664.036	671.926	669.276	5.597	0.883

Tabulka 10.B: Algoritmus „Nejdelší společná sekvence znaků dvou řetězců“, překladač clang/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
100	A = 25; B = 14	112.274	121.787	119.815	120.205	120.716	120.615	2.554	2.247
100	A = 27; B = 14	260.17	332.639	291.857	276.526	309.324	284.317	20.484	7.398
100	A = 29; B = 14	717.738	772.422	745.678	734.656	756.277	747.334	15.95	2.254

Tabulka 11.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Fibonacciho posloupnost“, překladač g++/c++20, sestava Notebook – Ubuntu.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 44	2.74	3.21	2.97	2.83	3.01	2.98	0.11	3.78
30	N = 45	4.73	4.94	4.84	4.77	4.91	4.85	0.06	1.36
30	N = 46	7.91	8.32	8.12	8.02	8.24	8.15	0.12	1.51

Tabulka 11.B: Algoritmus „Consteval – Fibonacciho posloupnost“, překladač g+/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 44	4.427	5.084	4.593	4.498	4.651	4.557	0.152	3.385
300	N = 45	7.199	7.574	7.367	7.293	7.438	7.346	0.116	1.608
300	N = 46	11.459	12.179	11.753	11.522	11.878	11.759	0.160	1.387

Tabulka 12.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Minimální, maximální a složená čísla v poli“, překladač clang/c++20, sestava MacBook – macOS.

Počet provedených překladů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
30	N = 4E+4	7.52	8.06	7.83	7.56	8.01	7.86	0.18	2.34
30	N = 5E+4	10.60	13.10	12.19	11.62	12.91	12.24	0.70	5.86
30	N = 6E+4	16.04	17.92	17.05	16.76	17.36	17.11	0.50	3.02

Tabulka 12.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Minimální, maximální a složená čísla v poli“, překladač clang/c++17, sestava MacBook – macOS.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 4E+4	0.260	0.277	0.262	0.260	0.262	0.261	0.003	1.242
300	N = 5E+4	0.396	0.402	0.398	0.397	0.399	0.397	0.001	0.445
300	N = 6E+4	0.561	0.57	0.565	0.562	0.567	0.564	0.002	0.416

Tabulka 13.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Evaluace determinantu matice“, překladač g++/c++20, sestava Notebook – Ubuntu.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	I = 2	6.03	6.97	6.44	6.23	6.71	6.39	0.25	4.06
30	I = 3	8.88	9.99	9.36	9.04	9.69	9.27	0.33	3.61
30	I = 4	12.01	14.80	13.04	12.18	13.78	12.78	0.82	6.43

Tabulka 13.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Evaluace determinantu matice“, překladač g++/c++17, sestava Notebook – Ubuntu.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	I = 2	0.286	0.313	0.296	0.294	0.300	0.296	0.006	2.250
300	I = 3	0.429	0.506	0.455	0.450	0.468	0.453	0.019	4.260
300	I = 4	0.571	0.653	0.594	0.581	0.607	0.586	0.019	3.395

Tabulka 14.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač g++/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 10	0.15	0.31	0.23	0.18	0.28	0.24	0.05	21.66
30	N = 11	1.12	1.39	1.24	1.14	1.31	1.24	0.08	6.76
30	N = 12	7.00	8.05	7.47	7.31	7.74	7.41	0.30	4.19

Tabulka 14.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 10	0.022	0.024	0.023	0.023	0.023	0.023	0.0003	1.443
300	N = 11	0.126	0.129	0.127	0.126	0.127	0.126	0.0008	0.653
300	N = 12	0.709	0.721	0.711	0.710	0.711	0.710	0.002	0.378

Tabulka 14.C: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač Clang/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 10	0.90	1.29	1.07	0.95	1.16	1.10	0.11	10.93
30	N = 11	5.71	6.24	5.92	5.78	6.09	5.87	0.16	2.90
30	N = 12	35.03	35.99	35.69	35.45	35.96	35.74	0.30	0.85

Tabulka 14.D: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač Clang/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 10	0.025	0.027	0.026	0.026	0.026	0.026	0.0003	1.343
300	N = 11	0.139	0.145	0.140	0.139	0.141	0.140	0.001	1.099
300	N = 12	0.823	0.861	0.827	0.824	0.827	0.825	0.006	0.838

Tabulka 14.E: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 10	0.73	0.86	0.79	0.75	0.82	0.80	0.03	4.80
30	N = 11	4.50	4.94	4.70	4.52	4.79	4.71	0.13	2.92
30	N = 12	27.01	28.90	28.35	28.15	28.81	28.56	0.57	2.05

Tabulka 14.F: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Počet řešení pro N-Queens problém“, překladač MSVC/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 10	0.022	0.039	0.024	0.023	0.024	0.023	0.003	13.972
300	N = 11	0.124	0.170	0.127	0.125	0.126	0.125	0.007	6.370
300	N = 12	0.720	0.749	0.728	0.725	0.729	0.726	0.004	0.659

Tabulka 15.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač g++/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 500	1.37	1.81	1.63	1.58	1.72	1.66	0.10	6.85
30	N = 600	2.67	2.99	2.87	2.76	2.96	2.90	0.10	3.66
30	N = 700	4.44	4.75	4.62	4.46	4.71	4.62	0.09	2.07

Tabulka 15.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 500	0.131	0.134	0.132	0.132	0.133	0.132	0.0008	0.67
300	N = 600	0.225	0.230	0.226	0.225	0.226	0.225	0.0009	0.415
300	N = 700	0.354	0.361	0.356	0.355	0.356	0.356	0.001	0.487

Tabulka 15.C: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač Clang/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 500	9.06	9.96	9.55	9.29	9.82	9.57	0.27	2.90
30	N = 600	16.11	18.94	17.26	16.42	17.99	16.98	0.85	5.04
30	N = 700	27.11	29.94	28.85	28.09	29.64	29.19	0.90	3.18

Tabulka 15.D: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač Clang/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 500	0.111	0.117	0.112	0.112	0.112	0.112	0.0009	0.863
300	N = 600	0.191	0.196	0.192	0.191	0.192	0.191	0.001	0.648
300	N = 700	0.301	0.307	0.302	0.301	0.302	0.302	0.001	0.596

Tabulka 15.E: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 500	7.64	8.38	8.03	7.71	8.22	8.08	0.24	3.06
30	N = 600	14.07	16.06	15.14	14.57	15.72	15.23	0.62	4.16
30	N = 700	23.01	27.49	24.75	23.61	25.94	24.28	1.34	5.52

Tabulka 15.F: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Consteval – Rychlé řazení prvků“, překladač MSVC/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 500	0.139	0.181	0.142	0.139	0.141	0.140	0.007	5.280
300	N = 600	0.236	0.275	0.240	0.238	0.239	0.238	0.006	2.857
300	N = 700	0.372	0.400	0.375	0.373	0.375	0.374	0.004	1.316

Tabulka 16.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač g++/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 29; K = 11	1.92	2.10	2.02	1.96	2.08	2.04	0.05	2.82
30	N = 30; K = 11	3.00	3.25	3.16	3.16	3.22	3.20	0.07	2.38
30	N = 31; K = 11	4.80	5.01	4.93	4.85	4.98	4.94	0.05	1.19

Tabulka 16.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 29; K = 11	0.186	0.191	0.187	0.186	0.187	0.186	0.001	0.562
300	N = 30; K = 11	0.294	0.299	0.295	0.294	0.295	0.295	0.001	0.500
300	N = 31; K = 11	0.453	0.486	0.459	0.455	0.459	0.456	0.008	1.810

Tabulka 16.C: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač Clang/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 29; K = 11	10.85	11.88	11.12	10.86	11.29	11.06	0.27	2.47
30	N = 30; K = 11	16.58	18.07	17.36	17.26	17.73	17.38	0.46	2.70
30	N = 31; K = 11	26.10	27.99	27.12	26.70	27.54	27.26	0.54	2.03

Tabulka 16.D: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač Clang/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 29; K = 11	0.192	0.197	0.193	0.193	0.194	0.193	0.0009	0.521
300	N = 30; K = 11	0.304	0.311	0.305	0.305	0.305	0.305	0.001	0.428
300	N = 31; K = 11	0.472	0.481	0.475	0.473	0.475	0.474	0.002	0.464

Tabulka 16.E: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 29; K = 11	7.61	8.54	8.09	7.70	8.37	8.22	0.30	3.82
30	N = 30; K = 11	13.10	14.30	13.76	13.30	14.09	13.79	0.38	2.81
30	N = 31; K = 11	21.08	22.97	22.04	21.65	22.65	21.91	0.59	2.74

Tabulka 16.F: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Hodnota binomického koeficientu“, překladač MSVC/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 29; K = 11	0.349	0.377	0.352	0.350	0.352	0.351	0.004	1.419
300	N = 30; K = 11	0.552	0.597	0.556	0.554	0.556	0.555	0.007	1.410
300	N = 31; K = 11	0.860	0.898	0.864	0.861	0.864	0.862	0.006	0.783

Tabulka 17.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač g++/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 20000	1.89	2.02	1.96	1.93	2.00	1.96	0.04	2.20
30	N = 27500	3.60	3.90	3.79	3.69	3.88	3.84	0.10	2.69
30	N = 35000	5.75	6.20	5.96	5.84	6.07	5.96	0.12	2.21

Tabulka 17.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 20000	0.486	0.496	0.490	0.488	0.490	0.489	0.002	0.422
300	N = 27500	0.912	0.949	0.919	0.914	0.917	0.914	0.011	1.238
300	N = 35000	1.465	1.477	1.470	1.467	1.473	1.47	0.002	0.197

Tabulka 17.C: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač Clang/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 20000	10.27	11.00	10.75	10.74	10.89	10.81	0.19	1.80
30	N = 27500	19.62	20.80	20.27	19.75	20.58	20.22	0.35	1.78
30	N = 35000	31.93	33.00	32.49	32.05	32.85	32.46	0.33	1.04

Tabulka 17.D: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač Clang/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 20000	0.528	0.539	0.531	0.530	0.531	0.531	0.002	0.482
300	N = 27500	0.989	1.029	0.999	0.991	1.000	0.993	0.013	1.394
300	N = 35000	1.591	1.652	1.600	1.594	1.602	1.595	0.013	0.866

Tabulka 17.E: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 20000	9.63	10.51	10.09	9.85	10.35	10.08	0.26	2.63
30	N = 27500	18.50	19.45	19.02	18.79	19.35	19.01	0.29	1.59
30	N = 35000	29.50	31.43	30.48	29.89	31.09	30.51	0.63	2.11

Tabulka 17.F: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Test prvočíselnosti“, překladač MSVC/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 20000	0.538	0.584	0.544	0.540	0.544	0.541	0.008	1.558
300	N = 27500	1.009	1.082	1.028	1.010	1.032	1.014	0.025	2.517
300	N = 35000	1.622	1.664	1.631	1.624	1.634	1.629	0.008	0.514

Tabulka 18.A: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač g++/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 2000	2.00	2.36	2.24	2.16	2.32	2.26	0.09	4.38
30	N = 3000	5.30	5.69	5.52	5.38	5.66	5.59	0.13	2.56
30	N = 4000	9.15	9.84	9.56	9.20	9.77	9.62	0.22	2.33

Tabulka 18.B: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač g++/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 2000	0.087	0.091	0.088	0.087	0.089	0.088	0.0009	1.101
300	N = 3000	0.196	0.201	0.197	0.197	0.197	0.197	0.001	0.519
300	N = 4000	0.349	0.357	0.351	0.350	0.351	0.350	0.001	0.523

Tabulka 18.C: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač Clang/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 2000	6.98	7.44	7.27	7.16	7.37	7.32	0.12	1.80
30	N = 3000	15.52	17.19	16.46	15.94	17.05	16.44	0.49	3.05
30	N = 4000	28.92	30.40	29.89	29.63	30.27	30.02	0.45	1.53

Tabulka 18.D: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač Clang/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 2000	0.083	0.090	0.085	0.083	0.086	0.084	0.002	2.596
300	N = 3000	0.187	0.192	0.188	0.188	0.188	0.188	0.001	0.564
300	N = 4000	0.333	0.340	0.334	0.333	0.334	0.333	0.001	0.471

Tabulka 18.E: Kompletní statistické veličiny získaných časů potřebných k překladu, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Parametr	Min. hodnota (min)	Max. hodnota (min)	Aritmetický průměr (min)	Dolní kvartil (min)	Horní kvartil (min)	Medián (min)	Směrodatná odchylka (min)	Variační koeficient (%)
30	N = 2000	5.41	5.70	5.58	5.47	5.67	5.60	0.09	1.67
30	N = 3000	13.52	14.50	14.06	14.15	14.31	14.15	0.30	2.18
30	N = 4000	22.51	25.97	24.35	23.64	25.20	24.54	1.04	4.34

Tabulka 18.F: Kompletní statistické veličiny vypočtené z časů potřebných k úplnému vykonání, dle nastaveného parametru. Algoritmus „Virtual constexpr – Variace řazení polí“, překladač MSVC/c++17, sestava PC – Windows.

Mohutnost množiny výstupů	Parametr	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
300	N = 2000	0.118	0.152	0.121	0.119	0.120	0.119	0.005	4.976
300	N = 3000	0.267	0.303	0.270	0.268	0.270	0.269	0.006	2.310
300	N = 4000	0.474	0.513	0.477	0.475	0.477	0.476	0.006	1.439

Tabulka 19.A: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač Clang/c++20, sestava MacBook – macOS.

Počet provedených překladů	Název souboru	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
30	Module1	0.570	0.650	0.611	0.600	0.620	0.610	0.016	2.707
30	Module2	0.870	0.960	0.935	0.920	0.950	0.940	0.018	2.000
30	Module3	0.770	0.830	0.801	0.780	0.810	0.800	0.012	1.607
30	Module4	0.830	0.910	0.884	0.877	0.890	0.880	0.015	1.798
30	Module5	2.760	2.920	2.810	2.780	2.822	2.805	0.031	1.153

Tabulka 19.B: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač Clang/c++17, sestava MacBook – macOS.

Počet provedených překladů	Název souboru	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
30	Header1	1.990	2.150	2.048	2.020	2.060	2.050	0.029	1.476
30	Header2	3.740	4.090	3.816	3.757	3.822	3.810	0.069	1.843
30	Header3	3.730	3.980	3.837	3.772	3.880	3.830	0.065	1.730
30	Header4	3.820	4.370	3.958	3.915	3.992	3.930	0.104	2.684
30	Header5	7.280	7.640	7.466	7.402	7.520	7.470	0.070	0.955

Tabulka 20.A: Velikost výstupu procesorů (konstantní hodnoty). Výstupy jsou referencí pro výsledky v tabulkách 20 a 23.

	Překladač			Překladač	
Soubor	Clang (KB)	g++ (KB)	Soubor	Clang (MB)	g++ (MB)
Module1	2.283	2.382	Header1	3.715	2.268
Module2	4.890	7.659	Header2	7.886	5.574
Module3	4.677	6.101	Header3	7.215	4.090
Module4	3.886	3.996	Header4	6.996	8.470
Module5	14.300	15.923	Header5	13.334	9.607

Tabulka 21.A: Kompletní statistické veličiny naměřené paměťové spotřeby během překladového procesu. Použitý překladač Clang/c++20, sestava MacBook – macOS.

Počet provedených překladů	Název souboru	Min. hodnota (MB)	Max. hodnota (MB)	Aritmetický průměr (MB)	Dolní kvartil (MB)	Horní kvartil (MB)	Medián (MB)	Směrodatná odchylka (MB)	Variační koeficient (%)
30	Module1	65.368	65.956	65.657	65.485	65.733	65.660	0.131	0.203
30	Module2	61.700	62.096	61.850	61.735	61.932	61.812	0.103	0.170
30	Module3	59.104	59.768	59.510	59.388	59.596	59.536	0.150	0.257
30	Module4	67.832	68.456	68.187	68.077	68.272	68.216	0.120	0.180
30	Module5	77.908	78.412	78.117	77.940	78.229	78.096	0.124	0.161

Tabulka 21.B: Kompletní statistické veličiny naměřené paměťové spotřeby během překladového procesu. Použitý překladač Clang/c++17, sestava MacBook – macOS.

Počet provedených překladů	Název souboru	Min. hodnota (MB)	Max. hodnota (MB)	Aritmetický průměr (MB)	Dolní kvartil (MB)	Horní kvartil (MB)	Medián (MB)	Směrodatná odchylka (MB)	Variační koeficient (%)
30	Header1	75.064	78.276	76.009	75.320	77.889	75.350	1.220	1.632
30	Header2	72.464	72.968	72.667	72.621	72.770	72.646	0.132	0.185
30	Header3	75.124	75.688	75.361	75.224	75.477	75.380	0.135	0.182
30	Header4	76.540	77.212	76.873	76.699	77.013	76.896	0.158	0.210
30	Header5	92.544	97.484	92.856	92.643	92.764	92.684	0.865	0.948

Tabulka 22.A: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač MSVC/c++20, sestava PC – Windows.

Počet provedených překladů	Název souboru	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Horní kvartil (ms)	Medián (ms)	Směrodatná odchylka (ms)	Variační koeficient (%)
30	Module1	232.00	512.00	270.90	242.00	257.00	248.00	70.87	26.61
30	Module2	323.00	584.00	364.56	341.75	367.75	352.50	46.93	13.09
30	Module3	284.00	517.00	313.60	298.00	318.25	304.00	40.72	13.21
30	Module4	350.00	648.00	415.36	368.00	397.25	375.50	88.09	21.57
30	Module5	286.00	535.00	361.70	292.50	477.75	311.00	94.34	26.53

Tabulka 22.B: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač MSVC/c++17, sestava PC – Windows.

Počet provedených překladů	Název souboru	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Horní kvartil (ms)	Medián (ms)	Směrodatná odchylka (ms)	Variační koeficient (%)
30	Header1	770.00	985.00	827.56	806.50	841.00	812.50	53.030	6.51
30	Header2	781.00	1188.00	905.90	849.00	975.00	870.00	90.20	10.12
30	Header3	698.00	1129.00	858.43	728.25	935.25	863.00	112.01	13.27
30	Header4	811.00	1295.00	892.00	833.25	901.25	852.50	99.35	11.33
30	Header5	942.00	1309.00	1047.16	962.25	1125.50	1003.50	98.72	9.58

Tabulka 24.A: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač g++/c++20, sestava Notebook – Ubuntu.

Počet provedených překladů	Název souboru	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
30	Module1	0.460	0.610	0.520	0.460	0.560	0.530	0.044	8.616
30	Module2	0.890	1.170	1.020	0.977	1.040	1.030	0.050	5.025
30	Module3	0.630	0.770	0.692	0.650	0.722	0.680	0.038	5.634
30	Module4	0.820	1.070	0.859	0.827	0.862	0.830	0.060	7.154
30	Module5	3.580	3.940	3.658	3.597	3.662	3.635	0.081	2.268

Tabulka 24.B: Kompletní statistické veličiny získaných časů potřebných k překladu. Použitý překladač g++/c++17, sestava Notebook – Ubuntu.

Počet provedených překladů	Název souboru	Min. hodnota (s)	Max. hodnota (s)	Aritmetický průměr (s)	Dolní kvartil (s)	Horní kvartil (s)	Medián (s)	Směrodatná odchylka (s)	Variační koeficient (%)
30	Header1	1.280	1.570	1.374	1.335	1.420	1.365	0.066	4.922
30	Header2	1.800	2.240	1.992	1.967	2.075	1.970	0.092	4.700
30	Header3	1.880	2.200	2.008	1.905	2.050	2.015	0.069	3.512
30	Header4	5.660	7.180	5.967	5.810	6.035	5.900	0.306	5.222
30	Header5	7.170	7.790	7.370	7.317	7.460	7.350	0.156	2.158

Tabulka 25.A: Kompletní statistické veličiny naměřené paměťové spotřeby během překladového procesu. Použitý překladač g++/c++20, sestava Notebook – Ubuntu.

Počet provedených překladů	Název souboru	Min. hodnota (MB)	Max. hodnota (MB)	Aritmetický průměr (MB)	Dolní kvartil (MB)	Horní kvartil (MB)	Medián (MB)	Směrodatná odchylka (MB)	Variační koeficient (%)
30	Module1	63.196	63.832	63.658	63.587	63.805	63.714	0.166	0.265
30	Module2	81.428	82.108	81.773	81.466	81.913	81.802	0.172	0.214
30	Module3	54.488	54.800	54.630	54.535	54.683	54.640	0.085	0.159
30	Module4	101.644	102.164	101.837	101.734	101.823	101.788	0.154	0.154
30	Module5	124.628	124.856	124.788	124.719	124.824	124.810	0.054	0.044

Tabulka 25.B: Kompletní statistické veličiny naměřené paměťové spotřeby během překladového procesu. Použitý překladač g++/c++17, sestava Notebook – Ubuntu.

Počet provedených překladů	Název souboru	Min. hodnota (MB)	Max. hodnota (MB)	Aritmetický průměr (MB)	Dolní kvartil (MB)	Horní kvartil (MB)	Medián (MB)	Směrodatná odchylka (MB)	Variační koeficient (%)
30	Header1	84.068	85.084	84.534	84.225	84.888	84.388	0.346	0.416
30	Header2	97.400	100.468	98.410	97.977	98.949	98.326	0.687	0.710
30	Header3	84.040	85.100	84.482	84.172	84.796	84.386	0.350	0.422
30	Header4	271.404	273.464	272.368	271.948	272.745	272.242	0.523	0.195
30	Header5	132.588	133.496	133.144	132.903	133.371	133.216	0.266	0.203