PD.TVZ.HR
POLYTECHNIC & DESIGN Vol. 10, No. 1, 2022.
Mrežna inačica: ISSN 2459-6302
Tiskana inačica: ISSN 1849-1995
DOI: 10.19279/TVZ.PD.2022-10-1-09

NOVITETI STANDARDA C++20

NOVELTIES OF THE C++20 STANDARD

Željko Kovačević, Aleksandar Stojanović
Tehničko veleučilište u Zagrebu, Vrbik 8, 10000 Zagreb, Hrvatska

SAŽETAK

Nakon velikog broja poboljšanja programskog jezika C++ dodanih u standardima C++11, C++14 i C++17 u prosincu 2020. godine objavljen je i standard C++20. Njegovom objavom programski jezik C++ dobio je nove mogućnosti i značajke koje dodatno poboljšavaju performanse C++ aplikacija i olakšavaju njihovo pisanje. Broj noviteta novog standarda poprilično je velik (veći nego u standardima C++14 i C++17) pa su u ovom radu opisani tek istaknutiji. Iako mnogi C++ prevoditelji u vrijeme pisanja ovog rada čak niti ne podržavaju standard C++ 20, svakako je važno istaknuti njegove prednosti što je prije moguće kako bi C++ programeri bili pravodobno obaviješteni o potencijalnim benefitima njihove uporabe.

Ključne riječi: *programiranje, C++, C++20, programski jezici, standard*

ABSTRACT

After a large number of improvements to the C++ programming language added in the C++11, C++14 and C++17 standards, the C++20 standard was published in December 2020. With its release, the C++ programming language has gained new features and characteristics that further improve the performance of C++ applications and make their writing easier. The number of novelties of the new standard is quite large (higher than in the standards C++ 14 and C++ 17), so in this paper only the more prominent ones are described. Although many C++ compilers at the time of writing this paper do not even support the C++ 20 standard, it is certainly important to point out its advantages as soon as possible so that C++ developers are informed in a timely manner about the potential benefits of their use.

Keywords: *programming, C++, C++20, programming languages, standard*

1. UVOD
1. INTRODUCTION

Programski jezik C++ prvobitno je počeo razvijati Bjarne Stroustrup 1979. s idejom proširivanja programskog jezika C. Početna inačica jezika zvana je "C s klasama" (eng. *C with classes*), da bi tek naknadno te godine počelo korištenje imena C++ [1]. Međutim, jezik je inicijalno standardiziran tek 1998. (standard C++98, ISO/IEC 14882:1998), a zatim je dobio manju nadogradnju 2003. (standard C++03, ISO/IEC 14882:2003). Daljnji razvoj jezika bio je usporen te se na objavu novog standarda čekalo sve do 2011. godine. [2]. Tijekom njegova razvoja, planirani naziv novog standarda bio je C++0x, ali budući da je završen i odobren 2011. naziv se promijenio u C++11 (ISO/IEC 14882:2011). U daljnjem razvoju programskog jezika C++, standardi C++14 (ISO/IEC 14882:2014) i C++17 (ISO/IEC 14882:2017) objavljeni su 2014. i 2017., a u prosincu 2020. konačno je objavljen i standard C++20 (ISO/IEC 14882:2020). Izuzev standarda C++03, koji je predstavljao manju nadogradnju jezika u obliku ispravljanja postojećih grešaka, svaki idući standard donosi i određene novitete.

Najveći broj noviteta dolazi sa standardom C++11 u kojemu su predstavljeni novi tipovi pametnih pokazivača, lambda funkcije, automatska dedukcija tipa varijabli, semantika prijenosa, biblioteka paralelnog programiranja, predlošci s neograničenim brojem argumenata (eng. *variadic templates*) itd. [3] [4]

C++14 uvodi mnoge druge, a zatim proširuje i neke od već postojećih C++11 značajki. Tako je uz automatsku dedukciju tipa varijabli sada moguća i automatska dedukcija tipa povratne vrijednosti funkcije. Uvode se generičke lambda funkcije s *auto* tipovima varijabli kao parametrima, omogućuje se kreiranje predložaka varijabli (u prethodnim inačicama C++a bili su podržani samo predlošci funkcija i klasa) [5], uključujući i mnoga druga proširenja standardne biblioteke poput onih opisanih u [6], [7] i [8]. C++17 nastavlja dalje s uvođenjem novih te poboljšavanjem već postojećih značajki, pa uvodi nove atribute u programski jezik C++ [9] te općenito omogućuje korištenje atributa u imenicima i numeratorima. Uvode se izrazi sažimanja (eng. *fold expressions*) koji uvelike pojednostavljuju rad s predlošcima koji koriste neograničeni broj argumenata [10], specifikacije iznimki postaju dio tipa funkcije [11], a sada C++ podržava i ugniježđene definicije imenika. Osim navedenih poboljšanja jezika, C++17 donosi i novitete u standardnoj biblioteci poput *std::string_view*, *std::optional*, *std::any* itd. Svi su navedeni standardi kroz dugi niz godina uvelike poboljšali kvalitetu programskog jezika C++ te privukli mnoge nove programere. Zbog toga je bilo logično za očekivati njegov daljnji razvoj i nadogradnju kroz standard C++20.

2. MODULI
2. MODULES

Moduli su jedna od najvažnijih značajki standarda C++20. Oni predstavljaju datoteke programskog koda koje se prevode samo jednom te zbog toga mogu značajno ubrzati prevođenje C++ aplikacija [12]. Primjerice, prilikom korištenja datoteka zaglavlja (.h) svaka datoteka izvornog koda koja koristi direktivu *#include* kopirat će u sebe sav sadržaj navedene *.h* datoteke. Ukoliko se jedna datoteka zaglavlja koristi u više datoteka izvornog koda, ovakav pristup duplicira programski kod i povećava vrijeme prevođenja. Štoviše, ukoliko se sadržaj datoteke zaglavlja promijeni to će također zahtijevati i ponovno prevođenje svih datoteka programskog koda koji ju koriste, time ponovno povećavajući vrijeme prevođenja. Zbog ovakvih i sličnih problema standard C++20 predstavio je module za koje se u budućnosti očekuje da će u potpunosti zamijeniti datoteke zaglavlja.

Primjer 1 *Modul 'pozdrav'*

Example 1 *Module 'pozdrav'*

```
// pozdrav.cppm
export module pozdrav;
export auto pozdravSvijete() {
    return "Pozdrav svijete!";
}
/// main.cpp
#include <iostream>
import pozdrav; // uvoz modula!
int main() {
    std::cout << pozdravSvijete() <<
'\n';
}
```

Primjer 1 prikazuje modul *pozdrav* te glavni program koji ga koristi. Za razliku od trenutnog pristupa gdje se u datotekama zaglavlja (.h) nalaze prototipovi i deklaracije, a u .cpp datotekama implementacija, modul može sadržavati sve u samo jednoj datoteci. Iz modula je dostupno samo ono što je označeno ključnom riječi *export* (primjerice, funkcija *auto pozdravSvijete*), dok modul može sadržavati i privatni dio koda koji nije dostupan za korištenje. Modul je izoliran i može sadržavati vlastite dijelove koda (primjerice, direktive i deklaracije) koje neće biti u sukobu s kodom iz drugih modula. Zbog toga redoslijed uključivanja modula nije bitan, dok je to i dalje postojeći problem prilikom korištenja datoteka zaglavlja gdje se ipak mora paziti na redoslijed njihovog uključivanja. C++ programeri u ovom trenutku ne moraju birati između datoteka zaglavlja i modula jer je dopušteno njihovo kombiniranje. Međutim, preporuka je da se u budućnosti koriste isključivo moduli kako bi se izbjegli postojeći problemi s datotekama zaglavlja.

3. KONCEPTI
3. CONCEPTS

Iako predlošci funkcija i klasa korištenjem generičkih tipova olakšavaju programiranje, C++20 i u ovom području donosi novitete. Pri pisanju predložaka sada je moguće definirati i ograničenja nad parametrima te time dodatno specijalizirati predloške. Imenovanjem takvog ograničenja stvara se koncept. Da bi demonstrirali koncept C++20 pretpostavimo da želimo napisati predložak funkcije *suma* koja zbraja bilo koja dva podatka koja se mogu pretvoriti u realni broj (Primjer 2).

Primjer 2 *Koncept 'RealniBroj' i njegova primjena'*

Example 2 *Concept 'RealniBroj' and its application*

```
#include <iostream>
// koncept - zahtjeva da se tip može pretvoriti u double
template<class T>
concept RealniBroj = requires (T tip) { (double)tip; };

// primjena koncepta
template<RealniBroj T1, RealniBroj T2>
auto suma(T1 a, T2 b) {
    return a + b;
}
// korisnički tip (bez operatora double) !
class Kompleksni {
public:
    double re, im;
    Kompleksni(double _re, double _im) : re(_re), im(_im) {}
};
int main() {
    std::cout << suma(2, 5) << '\n';
    std::cout << suma(2.5, 5) << '\n';
    std::cout << suma(Kompleksni(0, 0), Kompleksni(1, 1)) << '\n'; // greška!
}
```

Kreiran je koncept *RealniBroj* koji pomoću ključne riječi *requires* definira ograničenje generičkog tipa na način da ga mora biti moguće pretvoriti u tip *double*. U slučaju korisnički-definiranih tipova (klasa) to znači da mora postojati preopterećenje operatora *double*. Da bi ograničenje bilo primijenjeno, predložak funkcije *suma* koristi koncept *RealniBroj* kao tip parametra predloška, a prilikom njegova korištenja u funkciji *main* zadnja naredba će rezultirati greškom jer za objekte tipa *Kompleksni* nije definirana pretvorba u tip *double*. Primjena koncepta *RealniBroj* mogla je biti izvedena i korištenjem naredbe

```
template<class T1, class T2> requires
RealniBroj<T1> && RealniBroj<T2>
```

što ukazuje i na mogućnost kombiniranja različitih koncepata u istom predlošku.

4. RASPONI
4. RANGES

Za rad s rasponima elementa C++20 predstavlja biblioteku *Ranges*. Ona nije namijenjena da zamjeni već postojeće iteratore (*begin*, *end*, *rbegin*, *rend* itd.) već se može koristiti kao alternativa koja nudi ljepši i razumljiviji programski kod.

Prilikom rada s rasponima moguće je koristiti i poglede (eng. *views*), odnosno prilagođivače raspona (eng. *range adaptors*) poput *filter*, *drop*, *transform*, *take*, *split* itd. koji će bez mijenjanja izvornog seta podataka dati novi "pogled" na njihov sadržaj (Primjer 3).

Prikazani primjer započinje pozivom funkcije *sort* koja sada ne prima iteratore na početak i kraj vektora (*begin* i *end*) već je kao argument dovoljno predati samo naziv vektora. Ovakav programski kod puno je čitljiviji i pregledniji nego u prijašnjem C++ standardu, a pogotovo ako je riječ o sortiranju cijelog vektora, odnosno bez proizvoljno postavljenih raspona. U nastavku koda koriste se nizovi pogleda poput

- *drop* – iz raspona izbacuje N početnih elemenata
- *take* – u rasponu ostavlja samo N početnih elemenata
- *filter* – filtrira raspon po odabranom kriteriju
- *reverse* – elementi u rasponu se raspoređuju u obrnutom redoslijedu
- *transform* – svaki element u rasponu se transformira po odabranom kriteriju

Osim navedenih, postoje i mnogi drugi poput *take_while*, *drop_while*, *split*, *join*, *common* itd. [13]

Primjer 3 *Rasponi u standardu C++20*

Example 3 *C++20 standard ranges*

```
#include <algorithm>
#include <ranges>
#include <vector>

int main() {
    std::vector<int> vec{ 1,5,3,2,6,4 };
    std::ranges::sort(vec); // 123456

    auto res = vec
       | std::views::drop(2) // 3456
       | std::views::filter([](const auto& v) {return !(v % 2); }) //46
       | std::views::reverse //64
       | std::views::transform([](auto n) { return n * n; }) //3616
       | std::views::take(1); // 36
}
```

5. UVJETNE NAREDBE I OPERATOR "SPACESHIP"

5. CONDITIONAL STATEMENTS AND THE "SPACESHIP" OPERATOR

Dolaskom standarda C++20 uvjetne naredbe mogu koristiti atribute *[[likely]]* i *[[unlikely]]*, a za usporedbu podataka dodan je i operator "spaceship" <=>. Navedeni atributi služe kako bi prevoditelju dopustili dodatne optimizacije pretpostavljajući da je jedan tok izvršavanja vjerojatniji (*[[likely]]*) nego drugi (*[[unlikely]]*). Također, dodatni operator <=> po principu rada podsjeća na funkciju *strcmp* te može vratiti 3 vrijednosti: -1 (*prvi* < *drugi*), 0 (*prvi* == *drugi*) ili 1 (*prvi* > *drugi*).

Primjer 4 *Atributi i "spaceship" operator*

Example 4 *Attributes and the "spaceship" operator*

```
#include <iostream>

int main() {
    int start{0}, kraj{100};
    auto res = start <=> kraj; // "spaceship" operator

    if (res < 0) [[likely]]
        std::cout << "start < kraj" << std::endl;
    else [[unlikely]]
        if (res == 0)
            std::cout << "start == kraj" << std::endl;
        else
            if (res > 0)
                std::cout << "start > kraj" << std::endl;
}
```

Primjer 4 demonstrira slučaj gdje se za usporedbu dvaju podataka koristi operator <=>, pri čemu se pretpostavlja da je varijabla *start* manja od varijable *kraj*. Sukladno pretpostavci, na odgovarajućim mjestima korišteni su atributi *[[likely]]* i *[[unlikely]]* kako bi se prevoditelju omogućile dodatne optimizacije. Međutim, korištenje ovih atributa nije ograničeno samo na naredbu *if* već se mogu koristiti i kod naredbe *switch,* odnosno prilikom korištenja *case* izraza.

Kada se radi s korisnički definiranim tipovima (klasama) operator <=> podrazumijevano ne postoji. Ipak, moguće ga je preopteretiti ili od prevoditelja zatražiti generiranje njegove podrazumijevane implementacije (*auto X::operator<=>(const Y&) const = default;*). U tom slučaju prevoditelj će generirati i čak šest podrazumijevanih implementacija drugih potrebnih operatora (==, !=, <, >, <= i >=).

6. OSTALI NOVITETI
6. OTHER NOVELTIES

U prethodnom tekstu detaljnije su objašnjene tek neke od najistaknutijih značajki koje donosi standard C++20. Osim njih moguće je spomenuti i neke druge poput

- Korutine (eng. *coroutines*) koje se mogu koristiti kod asinkronih I/O operacija, kreiranja generatora, za lijeno računanje (eng. *lazy computations*) itd.
- Klazula hvatanja [=] u lambda funkciji implicitno više ne hvata pokazivač *this* već je njega sada potrebno hvatati eksplicitno.
- <chrono> podržava kalendare i vremenske zone
- Formatiranje teksta korištenjem *std::format*
- itd.

6.1. KORUTINE
6.1. COROUTINES

Mnogi moderni programski jezici kao što su Python, JavaScript i C# podržavaju rad s korutinama. Korutine su funkcije čije se izvršavanje može zaustaviti i kasnije nastaviti. One se često upotrebljavaju za implementaciju iteratora i generatora kao i za asinkrono programiranje.

U sljedećem primjeru (Primjer 5) prikazan je generator koji može generirati beskonačan niz brojeva počevši od 1. On se sastoji od nekoliko dijelova koji će biti postepeno i ukratko opisani.

Funkcija *initGen* koja sadrži glavni kôd za generiranje brojeva. Može se odmah primijetiti da, iako je njen povratni tip *Generator<int>,* ona nema naredbu *return*. To je zato jer se u njoj nalazi naredba *co_yield* zbog koje prevodilac zna da je ova funkcija korutina i nju tretira drugačije od „običnih“ funkcija. Naredba *co_yield* u svakoj iteraciji generira novu vrijednost i na mjestu svog izvršenja zaustavlja izvršavanje funkcije. Nakon što se ponovo pokrene, izvršavanje se nastavlja od naredbe koja slijedi *co_yield*. Isto tako, ova se funkcija naizgled izvršava u beskonačnoj petlji. Međutim, ovdje se ne radi o beskonačnom izvršavanju jer to je objekt koji se po potrebi može obrisati kada generiranje vrijednosti više nije potrebno, kao što to prikazuje Primjer 6.

Nakon pokretanja ovaj će program ispisati brojeve 1, 2, 3, 4 i 5. Nakon što je generator inicijaliziran i pridružen varijabli *generator*, petlja *for* će od njega tražiti iduću vrijednost pet puta. To se događa pozivom funkcije *iducaVrijednost* koja svaki put pokrene generator čime on nastavlja izvršavanje od mjesta na kojem je stao. Nakon pet iteracija izlazi se iz petlje, nakon čega će biti izvršen destruktor generatora (ako pretpostavimo da je ovaj kôd unutar funkcije). Time će generator biti uništen. Primjer 7 pokazuje ključni dio implementacije generatora - strukturu *Generator*.

Primjer 5 *Primjer generatora.*

Example 5 *Generator example.*

```
Generator<int> initGen() {
    int v = 1;
    while (true) {
        co_yield v;  // vrati v i zaustavi izvršavanje
        ++v;
    }
}
```

Primjer 6 *Primjer koda funkcije koja koristi generator.*

Example 6 *Sample code of a function using the generator.*

```
Generator<int> generator = initGen();  // inicijaliziraj generator
for (int i = 0; i < 5; ++i) {
    int val = generator.iducaVrijednost();  // ponovo pokreni izvršavanje
    std::cout << val << std::endl;
}
```

Primjer 7 *Tip generatora s tipom obećanja.*

Example 7 *Generator type with its promise type.*

```
template<typename T>
struct Generator {
    struct promise_type {
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        auto get_return_object() {
            return Generator{ ko_handle::from_promise(*this) };
        }
        std::suspend_always yield_value(int value) {
            vrijednost = value;
            return {};
        }
        void unhandled_exception() {
            std::exit(1);
        }
        T vrijednost;
    };
    using ko_handle = std::coroutine_handle<promise_type>;
    ko_handle korutina;
    Generator(ko_handle h) : korutina(h) {}
    ~Generator() {
        if (korutina) {
            korutina.destroy();
        }
    }
    T iducaVrijednost() {
        korutina.resume();  // nastavi s izvršavanjem
        return korutina.promise().vrijednost;
    }
};
```

Ona u sebi sadrži podstrukturu za objekt takozvanog *obećanja* (eng. *promise*) [14]. Taj objekt sadrži članske funkcije neophodne za izvršavanje korutine. Primjerice, kada korutina započinje izvršavanje prvo se kreira objekt za stanje korutine pa se u njega kopiraju parametri korutine. Tada se poziva konstruktor objekta obećanja, a nakon toga *get_return_object* pa *initial_suspend* te po povratku iz *initial_suspend* započinje izvršavanje tijela korutine. Objekt obećanja mora imati odgovarajuće sučelje da bi ga se moglo koristiti za izvršavanje korutine.

6.2. KLAUZULA HVATANJA [=, THIS]
6.2. CAPTURE CLAUSE [=, THIS]

Pokazivač *this* uvijek se hvatao kao referenca, čak i kod implicitne specifikacije [=].

C++20 omogućava eksplicitnu specifikaciju hvatanja pokazivača *this* kao referencu, kao što pokazuje Primjer 8.

Iako je oblik [=, this] redundantan, omogućava pisanje čitljivijeg koda.

6.3. BIBLIOTEKA CHRONO
6.3. CHRONO LIBRARY

Ova je biblioteka definirana u zaglavlju <*chrono*> i imenskom prostoru *std::chrono*. Za verziju C++20 dobila je kalendar i vremenske zone. Primjer 9 pokazuje postavljanje varijable *datum* na 19.8.2021., a varijable *posljednji_dan* na 31.8.2021., ali na način da se dan postavi na posljednji dan u mjesecu (kolovozu).

Primjer 8 *Primjer hvatanja pokazivača this.*

Example 8 *Example showing capture of the this pointer.*

```
struct Test {
    int n = 123;

    void f() {
        auto a = [=] {
            cout << n << endl;
        };  // kopiranje
        auto b = [=, *this] {
            cout << n << endl;
        };  // kopiranje
        auto c = [=, this] {
            cout << n << endl;
        };  // greška prije C++20; u C++20 isto što i [=]

        a();
        b();
        c();
    }
};
```

Primjer 9 *Postavljanje datuma s chrono bibliotekom.*

Example 9 *Setting a date variable with chrono library.*

```
#include <chrono>

int main() {
    using namespace std;
    using namespace std::chrono;

    year_month_day datum{ year(2021) / 8 / 19 };
    cout << datum << endl;  // ispisuje 2021-08-19

    year_month_day posljednji_dan{ year(2021) / (chrono::August / last) };
    cout << posljednji_dan << endl;  // ispisuje 2021-08-31
}
```

U biblioteci *chrono* datum se postavlja sintaksom koja podsjeća na konvencionalni način pisanja datuma, operatorom "/" kao što pokazuje Primjer 9 kod varijable *datum*.

6.4. FORMATIRANJE TEKSTA
6.4. TEXT FORMATTING

Formatiranje teksta još je jedan od mnogih noviteta verzije C++20.

Zaglavlje <*format*> sadrži nove funkcije za formatiranje teksta koje nadopunjuju postojeće slične funkcije kao što su *printf* i *I/O streams* biblioteka. Ova je biblioteka zasnovana na tri principa:

- Formatiranje pomoću rezerviranih mjesta (eng. *placeholder*)
- *Type-safe* formatiranje
- Formatiranje prilagođeno za korisničke tipove

Primjer 10 *Funkcija format.*

Example 10 *The format function.*

```
int main() {
    string s = format("{0}, {1}", "abc", "def");
    cout << s << endl;  // abc, def

    s = format("Broj = {:.2f}", 0.2971);  // dva decimalna mjesta
    cout << s << endl;  // Broj = 0.30

    s = format("Broj = {:.0f}", 1.95);
    cout << s << endl;  // Broj = 2
}
```

U nastavku je prikazano nekoliko jednostavnih primjera upotrebe funkcije *format*. Rezerviranim se mjestima ({0}, {1} itd.) specificira lokacija na kojoj će se umetnuti predani tekst. Nadalje, brojeve možemo formatirati tako da, primjerice, specificiramo s koliko se decimalnih mjesta ispisuju. Ovo pokazuje Primjer 10

Formatiranje ima puno više mogućnosti od onih koje su ovdje prikazane, kao što je prilagođavanje korisničkim tipovima, upisivanje formatiranog teksta u zadani tôk (eng. *stream*), razne mogućnosti formatiranja brojeva i znakovnih nizova itd.

7. ZAKLJUČAK
7. CONCLUSION

Kao jedan od najpopularnijih programskih jezika današnjice, C++ se neprestano razvija. Kroz redovite izlaske novih C++ standarda ovaj programski jezik proširuje se novim značajkama, poboljšava i pojednostavljuje te time opravdava status modernog programskog jezika koji prati aktualne trendove i potrebe zajednice. S obzirom na količinu i kvalitetu noviteta, standard C++20 zasigurno je pridonio tom dojmu i učinio C++ još boljim i učinkovitijim programskim jezikom. Međutim, čak i u trenutku pisanja ovog teksta nakon tek nedavno objavljenog standarda C++20, postoji i uređuje se nacrt sljedećeg C++ standarda (C++23) [15]. On će se, kao i njegovi prethodnici, usredotočiti na daljnje poboljšanje jezika i dodavanje novih značajki i algoritama u standardnu biblioteku. U daljnjem radu usredotočit ćemo se upravo na njega i njegovu primjenu.

8. REFERENCE
8. REFERENCES

[1.] B. Stroustrup, »When was C++ invented?,« 7 3 2010. [Mrežno]. Available: https://www.stroustrup.com/bs_faq.html#invention. [Pokušaj pristupa 8 8 2021].

[2.] H. Sutter, »We have an international standard: C++0x is unanimously approved,« 12 8 2011. [Mrežno]. Available: https://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/. [Pokušaj pristupa 2021 8 8].

[3.] N. M. Josuttis, The C++ Standard Library - A Tutorial and Reference, 2nd Edition, Addison Wesley Longman, 2012.

[4.] J. L. B. M. Stanley Lippman, C++ Primer (5th Edition), Addison-Wesley Professional, 2012.

[5.] G. D. Reis, »Variable Templates (Revision 1),« Texas A&M University, 2013.

[6.] M. Spertus, »Wording for Addressing Tuples by Type: Revision 2,« 19 4 2013. [Mrežno]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3670.html. [Pokušaj pristupa 2021 8 8].

[7.] D. V. H. B. Howard Hinnant, »Shared locking in C++,« 19 4 2013. [Mrežno]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3659.html. [Pokušaj pristupa 8 8 2021].

[8.] J. Yasskin, »exchange() utility function, revision 3,« 19 4 2013. [Mrežno]. Available: http://www.open-std.org/JTC1/sc22/WG21/docs/papers/2013/n3668.html. [Pokušaj pristupa 2021 8 8].

[9.] R. Smith, »Working Draft, Standard for Programming Language C++,« 6 2 2017. [Mrežno]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4640.pdf. [Pokušaj pristupa 2021 8 8].

[10.] »Folding expressions,« 7 11 2014. [Mrežno]. Available: https://isocpp.org/files/papers/n4295.html. [Pokušaj pristupa 2021 8 8].

[11.] J. Maurer, »P0012R1: Make exception specifications be part of the type system, version 5,« 22 10 2015. [Mrežno]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0012r1.html. [Pokušaj pristupa 2021 8 8].

[12.] R. Grimm, C++20, LeanPub, 2020.

[13.] B. Revzin, »C++20 Range Adaptors and Range Factories,« 28 2 2021. [Mrežno]. Available: https://brevzin.github.io/c++/2021/02/28/ranges-reference/. [Pokušaj pristupa 2021 8 8].

[14.] D. Friedman i D. Wise, »The Impact of Applicative Programming on Multiprocessing,« u International Conference on Parallel Processing, pp. 263–272., 1976.

[15.] T. Köppe, »Working Draft, Standard for Programming Language C++,« 18 6 2021. [Mrežno]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4892.pdf. [Pokušaj pristupa 2021 8 8].

AUTORI · *AUTHORS*

● **Željko Kovačević -** nepromjenjena biografija nalazi se u časopisu Polytechnic & Design Vol. 8, No. 3, 2020.

Korespondencija · *Correspondence*
zeljko.kovacevic@tvz.hr

● **Aleksandar Stojanović**
Predavač je na Tehničkom veleučilištu u Zagrebu i izvodi nastavu na predmetima iz područja programskog inženjerstva, objektno-orijentiranog programiranja i naprednog programiranja. Godine 1996. diplomirao je informacijske i komunikacijske znanosti na Filozofskom fakultetu sveučilišta u Zagrebu, a 1999. godine magistrirao je računarstvo u SAD-u na sveučilištu Midwestern State University te je radio kao programski inženjer u području telekomunikacija, financija i energetike. Godine 2019. doktorirao je na Filozofskom fakultetu sveučilišta u Zagrebu s disertacijom pod naslovom "Metoda automatske detekcije naglašenih riječi u zvučnom zapisu". Njegova područja interesa uključuju programske jezike, prevodioce i principe programiranja. Autor je knjige “Elementi računalnih programa s primjerima u Pythonu i Scali”.

Korespondencija · *Correspondence*
aleksandar.stojanovic@tvz.hr