Korutine u programskom jeziku C++

Zadravec, Matija

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matija Zadravec

KORUTINE U PROGRAMSKOM JEZIKU C++

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača, 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Ovaj rad posvećujem mojoj Anji i cijeloj obitelji uz zahvalu na podršci i strpljenju tijekom mojeg studiranja.

Sadržaj

Uvod

Asinkrono programiranje je paradigma programiranja koja se fokusira na izvršavanje zadatka bez obzira na to da li je prethodni zadatak završen. U klasičnom, ili sinkronom, programiranju, zadaci se izvršavaju jedan za drugim i često se čeka na završetak svakog zadatka prije nego što se prijeđe na sljedeći. Nasuprot tome, asinkrono programiranje omogućuje izvršavanje više zadataka istovremeno, bez čekanja na završetak svakog zadatka prije nastavka s izvršavanjem drugih zadataka. Asinkrono programiranje je ključno za razvoj aplikacija koje zahtijevaju efikasno upravljanje resursima, skalabilnost i brzu responzivnost, posebno u modernim web aplikacijama, mrežnim servisima i aplikacijama s grafičkim sučeljem.

Jedan od načina koji riješavaju asinkrone zadatke su korutine. One su programske konstrukcije koje omogućuju prekidanje izvršavanja jednog dijela koda kako bi se izvršio drugi dio, a zatim se izvršavanje može vratiti na prethodno prekinuti dio. Osim što se koriste za rukovanje asinkronim zadacima, korutine omogućuju efikasno upravljanje resursima. Korutine su široko korištene u asinkronom programiranju, gdje se zadaci izvršavaju neovisno jedan o drugom, bez čekanja na završetak svakog pojedinog zadatka prije nastavka.

Pitanje koje se nameće samo po sebi je: zašto bi netko prije koristio korutine u C++ u odnosu na druge programske jezike? C++ je poznat po svojoj brzini izvršavanja i niskoj razini apstrakcije, što ga čini pogodnim za aplikacije koje zahtijevaju visoku performansu. Korutine u C++ implementirane su pomoću niskostepenog mehanizma koji omogućava efikasno upravljanje resursima i smanjenje troškova prebacivanja između korutina. To može rezultirati boljim performansama u odnosu na korutine u drugim jezicima koji koriste višu razinu apstrakcije Osim toga, u C++ imamo veću kontrolu nad upravljanjem memorijom u usporedbi s drugim programskim jezicima koji koriste automatsko upravljanje memorijom.

Zbog dobrih značajka korutina u programskom jeziku C++ pokazat ćemo kako se one koriste i implementirat ćemo različite primjere da se može shvatiti što je sve moguće napraviti s njima. Nakon toga se ukratko upoznajemo s korutinama u drugim programskim jezicima kako bi istaknuli mane koje programski jezik C++ ima u podršci za korutine.

Poglavlje 1

Korutine u programiranju

Objasnimo prvotno pojam korutina. Korutina je određena generalizacija funkcije, odnosno funkcija s nekim dodatnim svojstvima. Prisjetimo se kako funkcioniraju obične funkcije u programskim jezicima. Funkcija se poziva u nekom dijelu pozivnog programa i nakon poziva funkcije se kontrola toka premjesti na tijelo funkcije. Nakon toga se izvršava niz naredbi u funkciji te nakon što se izvrše sve naredbe funkcija eventualno vrati neku vrijednost i vrati kontrolu toka na pozivni program. Bitan detalj je taj da se funkcija izvrši do kraja, tj. nisu ostale neke naredbe koji bi trebale biti naknadno izvršene. Nakon vraćanje kontrole toka izvršavanje pozivnog programa se nastavlja od poziva funkcija pa nadalje.

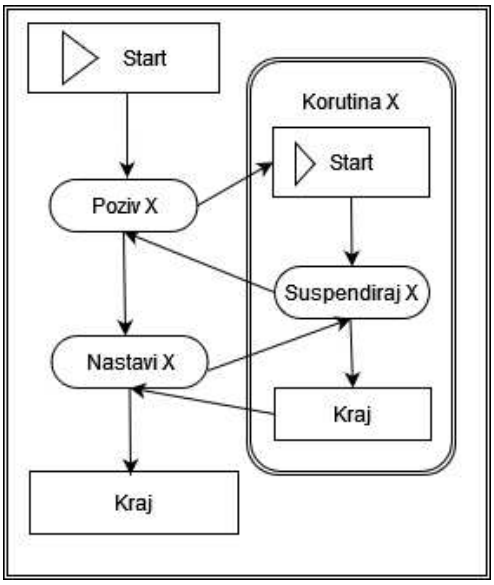
Kod korutina je situacija drugačija. Naime, korutinu možemo izvršavati po dijelovima. To znači da nakon niza izvršenih naredbi (ne svih naredbi) možemo suspendirati korutinu te se vratiti u pozivni dio programa. Točka u kojoj suspendiramo korutinu zove se točka suspenzije(eng. *suspend point*). Nakon povratka u pozivni dio programa izvršavamo naredbe u njemu te se vraćamo u korutinu u nekom željenom trenutku. Vraćanjem u tijelo korutine, izvršavanje se nastavlja od točke suspenzije. Postupak se ponavlja konačan broj puta. Korutina može vratiti vrijednost i ne mora, kao i svaka druga funkcija. Također, ona može vratiti vrijednost u svakoj točki suspenzije.

Za što uopće koristimo korutine? Korutine su jako korisne u asinkronom programiranju. Svrha korutina u asinkronom programiranju je pojednostavljivanje i poboljšanje čitljivosti koda koji se izvršava asinkrono. Tradicionalni pristupi asinkronom programiranju, kao što su callback funkcije, petlje događaja (eng. *event loops*) ili upotreba obećanja (eng. *promises*), mogu dovesti do piramide beskrajnih ugnježdenih callback poziva, što otežava čitanje i održavanje koda. Korutine nude elegantan i strukturiran pristup, a neki od razloga zašto se koriste su:

- Čitiljivost koda: korutine omogućuju pisanje asinkrone logike na način koji izgleda kao sinkroni kod. Koriste se ključne riječi `await` ili `co_await` koji čine kod sličnijim tradicionalnom, linearnom kodu

- Suspendiranje i nastavak: mogu suspendirati svoje izvršavanje zbog čega drugi dijelovi programa mogu obaviti svoj dio posla (korisno kada neki asinkroni zadatak čeka na I/O operaciju)
- Lakše rukovanje greškama: kroz korutine, greške se obrađuju putem izuzetaka, korištenjem try i catch blokova

te mnoge druge.



Slika 1.1: Princip rada jedne korutine

Osim korutina u C++, one su prisutne i u programskim jezicima kao što su Pyhton, JavaScript/TypeScript, C# (C Sharp) te mnogim drugima. Kako oni podržavaju korutine prokomentirat ćemo u zasebnom poglavlju.

1.1 Osnovne činjenice korutina u C++

Iako je prvo objašnjenje korutina bilo predstavljeno na početku druge polovice 20. stoljeća, podrška za korutine u C++-u se pojavila tek u verziji C++20. Podrška za korutine u C++ je dana kroz biblioteku `<coroutine>` no podrška nije potpuna nego sam programer mora napraviti (dobar) dio posla da se princip korutina može koristiti. Potpuna podrška za korutine u C++ se očekuje u sljedećoj verziji C++23 te bi tada korištenje korutina trebalo biti na apstraktnoj razini i korisnik neće trebati znati kako same korutine funkcioniraju u

pozadini. Za upotrebu korutina na apstraktniji način i na višoj razini u današnje vrijeme se također može koristiti biblioteka cppcoro koja je open source i implementirana je od strane Lewisa Bakera. Spomenuta biblioteka je od velikog značaja te će više o njoj biti objašnjeno u zasebnom poglavlju.

Ponovimo, svaka korutina je također i funkcija te se definira na sličan način kao i ona sama. Sama korutina se ne definira eksplicitno nego ju prepoznajemo ako sadrži određene ključne riječi. Za funkciju koja sadrži riječ `co_await`, `co_yield` ili `co_return` kažemo da je korutina. Ključna riječ `co_await` suspendira korutinu te vraća kontrolu toka na pozivni dio programa. Riječ `co_yield` ima istu zadaću kao `co_await` samo što, uz suspenziju, vraća privremenu povratnu vrijednost u pozivni program. Riječ `co_return` označava kraj korutine te može i ne mora vratiti vrijednost. Također, ona vraća sučelje za korutinu (eng. *coroutine interface*) kroz koji je omogućena komunikacija s pozivnim programom. Spomenuto sučelje je potrebno samostalno implementirati.

Pokažimo prvi primjer koji jednostavno i dobro prikazaju princip korutina.

```
1  #include <coroutine>
2  #include <iostream>
3  #include "task.h"
4
5  Task prva_korutina(){
6      std::cout<<"coro:  Prvi ulazak"<<std::endl;
7      co_await std::suspend_always();
8      std::cout<<"coro:  Drugi ulazak"<<std::endl;
9      co_await std::suspend_always();
10     std::cout<<"coro:  Treci ulazak"<<std::endl;
11 }
12
13 int main(){
14     Task pk_task = prva_korutina();
15     std::cout<<"main: Zavrsila inicijalizacija"<<std::endl;
16     pk_task.resume();
17     std::cout << "main: Prvi povratak" << std::endl;
18     pk_task.resume();
19     std::cout << "main: Drugi povratak" << std::endl;
20     pk_task.resume();
21     std::cout << "main: Kraj" << std::endl;
22
23     return 0;
24 }
```

Listing 1.1: Osnovni primjer korutine

Proučimo funkciju `prva_korutina()` u primjeru iznad. U linijama 7 i 9 primjećujemo ključnu riječ `co_await` što znači da je ta funkcija korutina te ona kao argument uzima *Awaiter* sučelje (koje će biti objašnjeno kasnije). Standardna C++ biblioteka nam daje

dva unaprijed definirana *Awaiter* sučelja: `std::supend_always` i `std::supend_never` gdje prvo sučelje suspendira korutinu dok kod drugog se izvršavanje korutine nastavlja bez vraćanja u pozivni dio programa. U glavnom dijelu programa je prvotno napravljena inicijalizacija korutine te kao inicijalizacijski tip ima kontrolni objekt *Task*. On mora imati implementiranu metodu `resume()` (koja se javlja u glavnom dijelu programa) s kojom se vraća kontrolu toka u korutinu.

Ispis gornjeg programa bi bio sljedeći:

```
main:  Zavrsila inicijalizacija
coro:  Prvi ulazak
main:  Prvi povratak
coro:  Drugi ulazak
main:  Drugi povratak
coro:  Treci ulazak
main:  Kraj
```

Dakle prvo se inicijalizira korutina bez pokretanja. Nakon toga se prvi put poziva metoda `resume()` s kojom pokrećemo korutinu te izvršava sve naredbe do ključne riječi `co_await` gdje dolazimo do točke suspenzije. U točki suspenzije se korutina suspendira i vraća u glavni dio programa te se izvršanje nastavi analogno. Nakon što se izvrše sve naredbe, korutina završava i uništava se kontrolni objekt. Završetak korutine se može dogoditi na dva načina: izvrše se sve naredbe i dođe do kraja funkcije ili dolazak do ključne riječi `co_return`.

Naš program također može imati više objekata sučelja iste korutine. Sučelje za korutinu (u ovom primjeru *Task*) ima spremljen atribut tipa `std::coroutine_handle` koji pamti stanje korutine u svakom trenutku od početka inicijalizacije objekta do poziva njegovog destruktora. To znači da možemo suspendirati naizmjence korutine i vraćati se u glavni dio programa bez da se zaboravi gdje je bila suspendirana pojedina korutina. Pogledajmo u sljedećem primjeru kako to funkcionira.

```
1  #include <coroutine>
2  #include <iostream>
3  #include "task.h"
4
5  Task prva_korutina(){
6      std::cout<<"coro:  Prvi ulazak"<<std::endl;
7      co_await std::suspend_always();
8      std::cout<<"coro:  Drugi ulazak"<<std::endl;
9      co_await std::suspend_always();
10     std::cout<<"coro:  Treci ulazak"<<std::endl;
11 }
12
```

```
13 int main(){
14     Task pk_task = prva_korutina();
15     Task pk_task_2 = prva_korutina();
16     std::cout<<"main: Zavrsila inicijalizacija"<<std::endl;
17     pk_task_2.resume();
18     pk_task.resume();
19     std::cout << "main: Prvi povratak" << std::endl;
20     pk_task.resume();
21     std::cout << "main: Drugi povratak" << std::endl;
22     pk_task.resume();
23     pk_task_2.resume();
24     std::cout << "main: Kraj" << std::endl;
25 
26     return 0;
27 }
```

Listing 1.2: Pokretanje iste korutine dva puta

Implementacija funkcije je identična kao i kod prvog primjera. Kod glavnog programa vidimo da na početku imamo inicijalizaciju 2 objekta *Task*: `pk_task` i `pk_task_2`. Ispis ovog programa bi bio sljedeći:

```
main:  Zavrsila inicijalizacija
coro:  Prvi ulazak
coro:  Prvi ulazak
main:  Prvi povratak
coro:  Drugi ulazak
main:  Drugi povratak
coro:  Treci ulazak
coro:  Drugi ulazak
main:  Kraj
```

Možemo zaključiti da svaki objekt korutinskog sučelja pamti stanje gdje je stao u korutini što dobro možemo vidjeti nakon drugog poziva metode `resume()` na objektu `pk_task_2`. Poslije tog poziva se dogodi ispis 'coro: Drugi ulazak'. Bitno je napomenuti da je svaki objekt uništen nakon završetka glavnog programa iako neka od korutina nije došla do svog kraja (kao u ovom primjeru `pk_task_2`).

Sljedeće što je zanimljivo kod korutine je da nema funkcijski stog. Nakon suspenzije se podaci, koji su potrebni za nastavak egzekucije, spremaju odvojeno od stoga i to nam dopušta da sekvencijalni kod izvršavamo asinkrono [1]. Lokacija koja je rezervirana za te podatke je memorija na hrpi (eng. *heap*). Pogledajmo sada što se događa u slučaju gdje se kod normalnih funkcija koristi funkcijski stog. Pokazat ćemo izvršavanje jednog jednostavnog programa gdje se jedna korutina poziva u drugoj.

```cpp
#include <coroutine>
#include <iostream>
#include "task.h"

Task prva_korutina(){
    std::cout<<"coro:  Prvi ulazak"<<std::endl;
    co_await std::suspend_always();
    std::cout<<"coro:  Drugi ulazak"<<std::endl;
}

Task druga_korutina(){
    std::cout << "Vanjska coro: Prvi ulazak" << std::endl;
    auto pk_task = prva_korutina();
    pk_task.resume();
    co_await std::suspend_always();
    std::cout << "Vanjska coro: Drugi ulazak" << std::endl;

}

int main(){
    Task pk_task = druga_korutina();
    std::cout<<"main: Zavrsila inicijalizacija"<<std::endl;
    pk_task.resume();
    std::cout << "main: Prvi povratak" << std::endl;
    pk_task.resume();
    std::cout << "main: Kraj" << std::endl;
    return 0;
}
```

Listing 1.3: Pokretanje korutine u korutini

Ispis danog programa je sljedeći:

```
main:  Zavrsila inicijalizacija
Vanjska coro:  Prvi ulazak
coro:  Prvi ulazak
main:  Prvi povratak
Vanjska coro:  Drugi ulazak
main:  Kraj
```

Slično kao i kod prijašnjih primjera, radili smo prvo inicijalizaciju vanjske korutine te na nju pozvali `resume()` metodu. Tada smo u toj korutini napravili inicijalizaciju nove korutine te na nju pozvali metodu za nastavak. Primjetimo da se unutarnja korutina ne izvrši do kraja jer nakon što se prvi put suspendira pozivni program (u ovom slučaju vanjska korutina) ne poziva više metodu za nastavak na kontrolnom objektu druge korutine.

Ovakvo izvršavanje koda je samo jedan od načina na koji bi se mogao izvršiti. Korutina se može izvršavati na način da metoda `resume()` u glavnom dijelu programa nastavlja unutarnju korutinu ili na neki drugi način. Kako implementirati tako nešto pokazat ćemo u idućim poglavljima kada se objasni *Awaiter* sučelje i promise tip za našu korutinu.

1.2 Dodatne klase i sučelja

Primjeri iz prošlog poglavlja se neće izvrišiti bez kontrolnog sučelja i promise tipa za našu korutinu. Programer je sam nužan implementirati spomenute dijelove te ih implementira na način kako on želi da njegova korutina funkcionira. Sljedeća potpoglavlja će ugrubo opisati kako se kontrolno sučelje i promise tip mogu implementirati. Nakon toga objasnit ćemo *Awaitables* i *Awaiter* sučelje koje nije potrebno implementirati, ali je dobar način za dodatnu manipulaciju korutina.

Kontrolno sučelje

Osnovni dio za korištenje korutina u C++ je kontrolno sučelje. Ono nam govori koji promise tip se koristi, kako se korutina ponaša u slučaju nastavka nje same (odnosno kako se ponaša u slučaju poziva `resume()` metode) i mnogo drugo. Kontrolni objekt za korutinu se stvara kod svakog poziva te pamti njeno stanje nakon suspenzije kako bi mogla nastaviti izvršavanje u trenutku vraćanja.

Pokazat ćemo jedan primjer kontrolne klase uz koju se mogu pokrenuti primjeri iz prošlog poglavlja (uz odgovarajući promise tip). Objasnit ćemo cijelu implementaciju klase, a u naknadnim poglavljima ćemo tu klasu proširivati kako bi iskoristili što veći potencijal korutina. Za sada ostanimo na korutinama koje ne vraćaju vrijednost.

```
1  #include <coroutine>
2  #include <iostream>
3  #include "promise.h"
4
5  class [[nodiscard]] Task{
6  public:
7      using promise_type = Promise;
8      Task(auto h) : cHandle{h} {
9      }
10     ~Task() {
11         if(cHandle)
12             cHandle.destroy();
13     }
14     Task(Task const &) = delete;
15
16     Task & operator=(Task const &) = delete;
```

```
17
18      Task(Task&& coro) noexcept {
19          cHandle = std::move(coro.cHandle);
20          coro.cHandle = nullptr;
21      }
22
23      Task& operator=(Task&& coro) noexcept {
24          if(&coro != this)
25          {
26              if(cHandle){
27                  cHandle.destroy();
28              }
29              cHandle = std::move(coro.cHandle);
30              coro.cHandle = nullptr;
31          }
32          return *this;
33      }
34
35      bool resume() const {
36          if(!cHandle)
37              return false;
38          cHandle.resume();
39          return !cHandle.done();
40      }
41  private:
42      std::coroutine_handle<promise_type> cHandle;
43  };
```

Listing 1.4: Kontrolno sučelje

Prvo što ćemo prokomentirati je atribut `cHandle` tipa `std::coroutine_handle<>`. On prima kao parametar promise tip koji je definiran za našu korutinu i klasa je iz standardne biblioteke `<coroutine>` te je ona sama zaslužna za spremanje stanja korutine. U sebi sadrži tri metode: `done()`, `destroy()` i `resume()`. Metode služe kako bi mogli manipulirati i kontrolirati korutinom kroz njeno izvršavanje. Objasnimo svaku od metoda kako bi znali kako one funkcioniraju i zašto su prisutne. Metoda `done()` vraća bool vrijednost te vraća `true` ako je korutina završila, a `false` dok to nije slučaj. Metoda `destroy()` se brine kako bi uništila `std::coroutine_handle<>` objekt na kraju izvršavanje korutine dok `resume()` nastavlja korutinu.

Metoda koju treba definirati je metoda s kojom ćemo nastavljati korutinu nakon suspenzije. U našem primjeru je to metoda `resume()`. Njezina implementacija je jednostavna i funkcionira na način da nakon provjere ako `std::coroutine_handle<>` nije prazan poziva na njega metodu `resume` i vraća `true` ako korutina nije završila. Konstruktor je jasan sam po sebi i nema neku posebnu funkcionalnost no destruktor vrijedi prokomentirati. Kada se poziva destruktor trebamo provjeriti postoji li još `cHandle`. U slučaju da

ne postoji (odnosno nije došlo do kraja izvršavanja korutine) trebamo uništiti `cHandle` s metododom `destroy()` da memorija na hrpi ne bi bila izgubljena.

Sljedeće oko čega treba obratiti pažnju su kopiranja i pridruživanja. Zadano se ponaša na način da bi kopiranje korutinskog sučelja kopiralo i `std::coroutine_handle` vrijednost, što bi moglo donijeti efekt da dva kontrolna objekta dijele istu korutinu.[4] Kad se radi u pridruživanju možemo primjetiti da se može dogoditi isti scenarij. Kako bi spriječili te scenarije biramo siguran način implementacije na način da ne dopuštamo niti kopiranje niti pridruživanje. Takva implementacija nas ograničava na neke načine te bi u nekim slučajevima bilo korisno da možemo koristiti premještanje odnosno da možemo spremiti korutinu u neki kontejner. Zbog tog razloga implementiramo move semantiku koju možemo vidjeti od 18. do 33. linije. U implementaciji pazimo da se iz prijašnjih kontrolnih sučelja obriše `cHandle` atribut da se nebi dogodio prije objašnjeni scenarij.

Zadnja stvar koja nije prokomentirana je atribut `[[nodiscard]]` koji je stavljen uz naziv klase. On nam služi za otkrivanje korutina koje su inicijalizirane, a nisu pokrenute. Ako postoji takva kompajler će javiti upozoranje za takvo postojanje.

Promise klasa

Uz sučelje za korutinu, obavezno je implementirati Promise klasu. To nam je zadnji potrebni kotačić za pokretanje programa iz prethodnih poglavlja. Spomenuta klasa se ugnijezdi unutar kontrolnog sučelja (primjer 1.5) i koristi se samo za korutinu. Njene osnovne zadaće su da prenese sve bitne podatke iz korutine u pozivni program i da ju konfigurira na određen način. Pod konfiguracijom korutine podrazumijevamo da odlučuje o suspenziji korutine kod inicijalizacije, vraćanje vrijednosti iz korutine, riješavanje iznimki u slučaju njihovog javljanja i mnoge druge. Kontrolno sučelje pristupa svim podacima i funkcijama iz Promise klase preko `std::coroutine_handle` objekta koji je dan kao atribut sučelja.

```
1  #include <coroutine>
2  #include <iostream>
3
4  struct Promise{
5      auto get_return_object(){
6          return std::coroutine_handle<Promise>::from_promise(*this);
7      }
8      auto initial_suspend(){
9          return std::suspend_always{};
10     }
11     auto final_suspend() noexcept {
12         return std::suspend_always{};
13
14     }
15     void unhandled_exception(){
16         std::terminate();
```

```
17      }
18      void return_void(){
19      }
20  };
```

Listing 1.5: Promise tip

Opišimo sada pobliže implementaciju klase Promise. U njoj imamo neke funkcije koje trebaju biti implementirane dok se neke mogu, ali ne moraju implementirati. Troje obaveznih metoda koja svaka Promise klasa ima su:

- `get_return_object()` - služi za kreiranje kontrolnog objekta korutine te mora vratiti coroutine_handle<>tip
- `initial_suspend()` – određuje kako se korutina ponaša kod inicijalizacije odnosno hoće li odmah krenuti izvršavanje do prvog suspendiranja ili će čekati resume od pozivnog programa
- `unhandled_exception()` - definira ponašanje u slučaju pojave iznimke
- `final_suspend()` – hoće li korutina biti suspendirana nakon što je završila izvršavanje svog tijela

Uz primjer 1.5 konačno se mogu pokrenuti programi iz prošlog poglavlja. Objasnimo sada utjecaj implementiranih metoda na prethodne primjere. U `get_return_object` metodi nam statička metoda `from_promise` napravi iz našeg Promise objekta traženi kontrolni objekt. Metoda `initial_suspend` je definirana na lijeni način jer se nakon inicijalizacije odmah suspendira (vraća `std::suspend_always`). Ako želimo da se korutina izvršava kod inicijalizacije potrebno je da `initial_suspend` vraća `std::suspend_never`. Kod `final_suspend` metode obično nemamo puno slobode. Ona bi uvijek trebala vraćati `std::suspend_always` jer korutina mora biti suspendirana kako bi mogla biti uništena. Metoda `unhandled_exception` je definirana također na jednostavan način te nema posebnu funkcionalnost. Sljedeće što je obavezno implementirati je jedna od sljedećih funkcija: `return_void`, `return_value`, `yield_value`. Prvu implementiramo ako u korutini koristimo `co_await` ili `co_return` bez argumenata funkcije. Druga je obavezna kad vraćamo neku vrijednost s `co_return` dok treća kad vraćamo vrijednost s `co_yield`. U našim primjerima do sada je bilo dovoljno implementirati `return_void` jer smo samo koristili `co_await`. Primjere implementacije funkcije koje su potrebne za vraćanje vrijednosti ćemo detaljno vidjeti u sljedećim poglavljima.

Pored obaveznih metoda, u Promise klasi mogu biti i metode koje su opcionalne. One su prisutne kako bi definirala neka specijalna ponašanja korutine kako se ne bi ponašala na uobičajen način. Radi se o metodama: `await_transform()`, operatorima `new()` i `delete()` te metodi `get_return_object_on_allocation_failure()`. Prva metoda

služi kako bi vrijednost `co_await` operator, ako nije klasični *Awaiter*, pretvorila u neki *Awaiter*. Ostale tri metode služe za kontrolu alokacije memorije i rješavanje iznimaka ako se dogodi kod alokacije. Već smo ranije spomenuli da korutina nema funkcijski stog te se stanje korutine sprema u memoriji na hrpi (eng. *heap*). Operatori `new()` i `delete()` se definiraju ako programer nije zadovoljan kako se alocira memorija na hrpi te implementira svoj način.

Awaitables i Awaiters

Još jedna stvar koja nije nužna, ali korutina može imati, je *Awaitables* sučelje sa svojim *Awaiter* sučeljima. Prvo ćemo razjasniti koja je razlika između *Awaitables* i *Awaiter* sučelja. *Awaitables* objekti su objekti koje metode `co_await` i `co_yield` prima kao argument te definiraju njihovo ponašanje i povratne tipove dok je *Awaiter* samo jedan poseban način na koji se može implementirati *Awaitable*. Oni služe za to da odrede ponašanje programa neposredno prije ili nakon suspenzije korutine.

Za početak ćemo pojasniti *Awaiter* jer je jednostavniji. Jednostavniji je iz tog razloga zato što za njega treba implementirati tri metode: `await_ready()`, `await_suspend()` i `await_resume()`. Objasnimo redom kada se pozivaju spomenute metode nakon što naiđemo na ključnu riječ `co_await`. Metoda `await_ready()` se poziva trenutak prije suspenzije korutine. Ona je ta koja određuje hoće li se korutina suspendirati ili će zanemariti suspenziju te nastaviti dalje. Informaciju nam daje kroz povratnu bool vrijednost. Vrijedi spomenuti da u ovoj metodi se nebi smjele pozivati `resume()` i `destroy()` funkcije jer korutina u ovom trenutku nije suspendirana. Metoda `await_suspend` se poziva odmah nakon što se korutina suspendirala te prima `std::coroutine_handle<Promise>` ili `std::coroutine_handle<void>` (u ovom drugom slučaju se ne može pristupiti promise objektu) kao argument funkcije. Povratni tip joj može biti `void` ili bool te određuje hoće li vratiti kontrolu toka na pogram iz koje je pozvana korutina ili ne. Ako je povratni tip `void` kontrola toka se vraća u pozivni dio progama u svakom slučaju dok u slučaju bool varijante se ona vraća samo ako je metoda vratila `true`. U slučaju `false` korutina se ponaša kao da nije ni došla do `co_await` te nastavlja sa svojim izvršavanjem. Postoji mogućnost da vrati `std::coroutine_handle<>` objekt kako bi nastavila izvršavanje neke druge korutine no takvo ponašanje nećemo opisivati ovdje. Zadnja metoda koju treba implementirati je `await_resume()`. Ona se poziva u slučaju da je suspenzija bila uspješna te služi za vraćanje vrijednosti ako je to potrebno.

Dva primjera Awaiters sučelja, `std::supend_always` i `std::suspend_never`, nudi standardna biblioteka i pokazana je njihova upotreba. U sljedećem primjeru pogledajmo jedan osnovni *Awaiter*.

```
1 #include <iostream>
2 #include <coroutine>
3 #include <promise.h>
```

```
4
5  class Awaiter{
6  public:
7      bool await_ready() const noexcept{
8          return true; //poziva netom prije suspenzije
9      }
10
11     void await_suspend(std::coroutine_handle<Promise> cHndl) const
       noexcept{
12         //poziva odmah nakon suspenzije
13     }
14
15     void await_resume() const noexcept{
16         //poziva nakon nastavka korutine
17     }
18
19 }
```

Listing 1.6: Primjer klase koja implementira *Awaiter* sučelje

Funkcija `await_ready()` vraća uvijek `true` dok ostale ne rade ništa. Zaključujemo da ovaj jednostavan *Awaiter* ima istu funkcionalnost kao i `std::suspend_never` jer zbog konstantnog vraćanja vrijednosti `true` korutina se nikad neće suspendirati. Da bi *Awaiter* iz gornjeg primjera imao identičnu funkcionalnost kao i `std::suspend_always` treba promjeniti `await_ready()` metodu na način da uvijek vraća false vrijednost. U praksi većinom funkcija `await_ready()` vraća `false` vrijednost jer u protivnom se suspenzija nikad ne bi dogodila.

1.3 Korutine s povratnom vrijednosti

U ovom poglavlju ćemo se fokusirati na korutine povratnih vrijednosti. Za razliku od prijašnjih primjera, u ovim će, nakon suspenzije, korutina vratiti vrijednost. Ključne riječi u ovom poglavlju su `co_yield` i `co_return` gdje uz pomoć prve možemo vratiti vrijednost nakon suspenzije dok druga može vratiti na kraju izvršavanja korutine. Cilj ovog poglavlja je, osim pokazati primjene korutina s vraćanjem vrijednosti, prilagoditi kontrolno sučelje i promise tip kako bi funkcionirala takva korutina.

```
1  #include <iostream>
2  #include <coroutine>
3  #include "generator.h"
4
5  Generator generator(int n){
6      int pot = 1;
7      for(int i = 0; i < n; ++i){
8          pot *= 2;
```

```
9           std::cout << "Coro: vracam se u pozivni dio programa" << std::
    endl;
10          co_yield pot;
11      }
12  }
13
14  int main(){
15      auto gen = generator(5);
16      while(gen.next())
17      {
18          std::cout << "Vratio sam vrijednost: " << gen.getValue() << std
    ::endl;
19      }
20
21      return 0;
22  }
```

Listing 1.7: Korutina s povratnom vrijednošću

U dosadašnjim primjerima smo imali samo korutine bez argumenata funkcije. Kao i kod svake druge funkcije možemo prosljeđivati argumente, ali postoje neke restrikcije. Naime, nailazimo na problem kada kroz argument funkcije prosljedimo referencu. Postojanje korutine traje duže nego život jednog poziva funkcije (poziva korutine do prvog suspendiranja). Zbog toga može doći do nesporazuma ako više puta inicijaliziramo korutinu s različitom vrijednosti koja je prosljeđena kao referenca. U nekim slučajevima će takav kod raditi kao što programer očekuje jer različiti kompajleri se ne ponašaju uvijek isto. Kako ne možemo znati uvijek ponašanje kompajlera i na koji način bi se kod izvršio, preporuka je da se ne koriste reference u deklariranju argumenta funkcije.

co_yield

Vratimo se sada primjeru 1.7 u kojoj korutina vraća vrijednost. U ovom poglavlju ćemo se više puta susresti s različitim generatorima jer uz pomoć korutina možemo generirati svakakve nizove brojeva. Ovaj generator je jednostavni generator potencije broja 2. Promotrimo prvo ispis danog primjera te prokomentirajmo tada njegovo izvršavanje:

```
Coro:  vracam se u pozivni dio programa
Vratio sam vrijednost:  2
Coro:  vracam se u pozivni dio programa
Vratio sam vrijednost:  4
Coro:  vracam se u pozivni dio programa
Vratio sam vrijednost:  8
Coro:  vracam se u pozivni dio programa
```

```
Vratio sam vrijednost:  16
Coro:  vracam se u pozivni dio programa
Vratio sam vrijednost:  32
```

Glavni program počinje s inicijalizacijom korutine kao i prijašnji primjeri te je također implementiran na lijeni način (korutina se ne pokreće odmah nakon inicijalizacije nego nakon prvog nastavljanja). Kao argument funkcije prima tip `integer` koji određuje koliko potencija broja dva treba ispisati. Umjesto naziva `resume()` koristimo `next()` jer je deskriptivnije u slučaju generatora, ali uloga im je identična. U ovom primjeru vidimo korisnost vraćanja `bool` vrijednosti u metodi `next()` jer na ovaj način se petlja izvršava tako dugo dok generator ne izvrši svoje generiranje do kraja. Metoda `next()` ne vraća vrijednost iz naše korutine nego za to koristimo funkciju `getValue()`. Od novih stvari u korutini pojavljuje se `co_yield` koji služi da vrati vrijednost u glavni dio programa koju onda dohvaćamo s metodom `getValue()`.

```
1 class Promise{
2         int stored_value;
3         ...
4         auto yield_value(int val){
5             stored_value = val;
6             return std::suspend_always{};
7         }
8     };
```

Listing 1.8: Nadogradnja *Promise* klase

Za korištenje ključne riječi `co_yield` trebamo implementirati metodu `yield_value()` u *Promise* klasi. U 1.8 je prikazan jedan način te implementacije. U klasi *Promise* imamo atribut `stored_value` koji je zadužen za spremanje trenutne vrijednosti korutine te ga uz funkciju `yield_value()` spremamo u objekt. Funkcija se poziva nakon poziva `co_yield` operatora. Također se iz `yield_value()` metode vidi da nakon `co_yield` suspendiramo korutinu. Ako želimo nekakvu posebnu logiku i izvršavanje korutine nakon `co_yield` ovdje je mjesto za tu implementaciju. Implementacija ostalih obaveznih funkcija u *Promise* klasi je jednak kao u 1.5

Zadnju stvar koju trebamo dodati je metoda u sučelju za korutinu za vraćanje vrijednosti. Njezina implementacija je prikazana na 1.9. Logika koja se krije iza nje je samo dohvaćanje vrijednosti iz svog vlastitog promise tipa. Ostatak klase je implementiran na jednak način kao u 1.4 primjeru.

```
1 class [[nodiscard]] Generator{
2 private:
3     std::coroutine_handle<Promise> mHandler;
4 public:
5     ...
```

```
    int getValue(){
        return mHandler.promise().stored_value;
        }
};
```

Listing 1.9: Nadogradnja sučelja za korutinu

Sučelje za korutinu nam može poslužiti da iterariramo kroz vrijednosti koje su vraćene kao povratne vrijednosti korutine. Kao motivaciju dajemo primjer generatora Fibonaccijevih brojeva i iteriranje kroz njih.

```
#include <iostream>
#include <coroutine>
#include "generator.h"

Generator<int> fibonacciGenerator(int n) {
    int a = 0, b = 1;
    for (int i = 0; i < n; ++i) {
        co_yield a;
        int temp = a;
        a = b;
        b += temp;
    }
}

int main() {
    Generator<int> fibGen = fibonacciGenerator(10);

    // Iteriranje kroz generirane Fibonacci brojeve
    for (const auto& value : fibGen) {
        std::cout << value << " ";
    }

    return 0;
}
```

Listing 1.10: Generator Fibonnacijevih brojeva

Kod 1.10 ispisuje 10 Fibonnacijevih brojeva:

```
0 1 1 2 3 5 8 13 21 34
```

Sljedeća zadaća je proširiti sučelje za korutinu na način da bi se kod 1.10 uspješno izvršavao. Da bismo omogućili iteraciju kroz objekte neke klase u C++, trebamo implementirati određene metode i sučelje koje C++ standardna biblioteka očekuje od objekta koji je iterabilan. Glavna sučelja za iteraciju u C++ su `begin()` i `end()`. Pošto naša

klasa ima posebnu logiku za iteriranje kroz svoje elemente, trebamo implementirati vlastiti iterator. Svaki iterator ima tri metode:

- `operator*`: Vraća referencu na trenutni element na koji iterator pokazuje.
- `operator++`: Pomiče iterator na sljedeći element u kolekciji.
- `operator==`: Provjerava jesu li dva iteratora različiti.

```
template <typename T>
struct [[nodiscard]] Generator{
    ...
    struct iterator{
        std::coroutine_handle<promise_type> handle;
        iterator(auto h) : handle{h} {}
        void next(){  // Pomocna rutina koju koristimo u ++ i begin()
            if(handle)
                handle.resume();
            if(handle.done())
                handle = nullptr;
        }
        iterator & operator++(){
            next();
            return *this;
        }
        T operator*() const{
            assert(handle != nullptr);
            return handle.promise().current_value;
        }
        bool operator==(iterator const &) const = default;
    };

    iterator begin() const {
        if(!mHandler || mHandler.done())
          return iterator{nullptr};
        iterator it{mHandler};
        it.next();
        return it;
    }
    iterator end() const {
        return iterator{nullptr};
    }

    std::coroutine_handle<promise_type> mHandler;
};
```

Listing 1.11: Sučelje za korutinu s iteratorom

Promotrima sad naš `Generator` s iteratorom i sučeljem koje je potrebno napraviti za iteriranje. `Generator` je napravljen kao predložak klase pa se može koristiti za više tipova i kod različitih primjena. Iterator kao atribut drži `std::coroutine_handle` tip uz kojeg možemo iterirati kroz elemente. Uz navedene metode, dodana je pomoćna metoda `next()` i konstruktor. Metoda `next()` samo poziva `resume()` na atribut iz klase te je od pomoći kod implementacije ostalih potrebih metoda. Operator uspoređivanja je uobičajen i nema posebnu implementaciju. Kod `operator*` pristupamo atributu iz *Promise* klase i vraćamo tu vrijednost, a `operator++` uzima vrijednost koja se vraća kod sljedeće točke suspenzije u korutini. Implementacija ostalih funkcija je opet jednaka kao i prije.

Pokazali smo do sada par primjera gdje korutina vraća vrijednosti u pozivni program. Sada želimo da nam pozivni program vrati vrijednost u korutinu. Za to će nam, uz sučelje za korutinu i promise tip, trebati odgovarajuća *Awaiter* klasa koja će napraviti dio posla. Pogledajmo jedan primjer korutine kao motivaciju gdje pozivni program vraća vrijednost u korutinu, a nakon toga objasnit ćemo potrebne adaptacije kod sučelja za korutinu, odgovarajućeg promise tipa i nove *Awaiter* klase.

```
#include <iostream>
#include "generator.h"

Generator coroExample(int max) {
    std::cout << "Coroutine start\n";

    for (int val = 1; val <= max; ++val) {
        std::cout << "Coroutine value: " << val << '\n';

        auto response = co_yield val;

        if (response == "PARAN") {
            std::cout << "Broj je paran\n";
        }
        else {
            std::cout << "Broj je neparan\n";
        }
    }

    std::cout << "Kraj korutine\n";
}

int main() {
    auto cExample = coroExample(3);

    std::cout << "**** Korutina pocinje\n";

    std::cout << "\n**** Nastavi korutinu\n";
    while (cExample.resume()) {
```

```
30          int value = cExample.getValue();
31          std::cout << "**** Korutina je suspendirana s vrijednosti: " <<
    value << '\n';
32
33          std::string back;
34          if (value % 2 == 0) {
35              back = "PARAN";
36          } else {
37              back = "NEPARAN";
38          }
39
40          std::cout << "\n**** Nastavi korutinu s vrijednosti: " << back
    << '\n';
41          cExample.setBackValue(back);
42      }
43
44      std::cout << "**** Korutina gotova\n";
45
46      return 0;
47  }
```

Listing 1.12: Vraćanje vrijednosti u korutinu

Korutina u primjeru je jednostavni generator koji ispisuje redom brojeve nakon svake suspenzije. U glavnom dijelu programa provjeravamo parnost broja te prosljeđujemo rezultate natrag do korutine gdje rezultat ispisujemo. Promotrimo njegov ispis:

```
**** Korutina pocinje

**** Nastavi korutinu
Start korutine
Vrijednost korutine:  1
**** Korutina je suspendirana s vrijednosti:  1

**** Nastavi korutinu
Broj je neparan
Vrijednost korutine:  2
**** Korutina je suspendirana s vrijednosti:  2

**** Nastavi korutinu
Broj je paran
Vrijednost korutine:  3
**** Korutina je suspendirana s vrijednosti:  3
```

```
**** Nastavi korutinu
Broj je neparan
Kraj korutine
**** Korutina gotova
```

Objasnimo jednu iteraciju danog primjera. U `while` petlji se nastavlja prvi put korutina te vraća vrijednost u glavni dio programa preko `co_yield` operatora kao u prošlim primjerima. Jedina razlika je što ovdje ima povratnu vrijednost tipa string. Kasnije ćemo objasnit zašto je to tako kada objasnimo dodatne klase. Nako što se vratimo u glavni dio programa, vrijednost koju želimo imati u korutinu vraćamo preko metode `setBackValue()` te ćemo tu vrijednost imati u varijabli `response` nakon sljedećeg nastavka korutine.

```
template<typename Handle>
class BackAwaiter {
  Handle cHandle = nullptr;
public:
  BackAwaiter() = default;

  bool await_ready() const noexcept {
    return false;
  }

  void await_suspend(Handle hdl) noexcept {
    cHandle = hdl;
  }

  auto await_resume() const noexcept {
    return cHandle.promise().back_value;
  }
};
```

Listing 1.13: *Awaiter* za vraćanje vrijednosti

Za egzekuciju 1.12 potreban nam je *Awaiter* koji sprema potrebnu vrijednost kod suspenzije. Kod 1.13 prikazuje potrebne modifikacije kod implementacije sučelja *Awaiter*. Kod suspenzije spremamo objekt koji upravlja korutinom (metoda `await_suspend()`) kako bi onda kod nastavljanja korutine mogli spremiti tu vrijednost u `back_value` varijablu *Promise* objekta našeg sučelja za korutinu (metoda `await_suspend()`). Vidjeli smo prije da je *Promise* objekt taj koji služi za spremanje podataka kod komunikacije iz sučelja u pozivni program, a sada vidimo da se to pojavljuje i kod obrnute komunikacije.

```
#include "back_awaiter.h"
#include <coroutine>
#include <string>

class [[nodiscard]] Generator {
```

```
6  public:
7    struct promise_type;
8  private:
9    std::coroutine_handle<promise_type> hdl;
10
11 public:
12   struct promise_type {
13     int current_value = 0;
14     std::string back_value;
15
16     ...
17
18     auto yield_value(int val) {
19       current_value = val;
20       back_value.clear();
21       return BackAwaiter<std::coroutine_handle<promise_type>>{};
22     }
23   };
24
25   ...
26
27   void setBackValue(const auto& val) {
28     hdl.promise().back_value = val;
29   }
30 };
```

Listing 1.14: Promjenjeni dio sučelja za korutinu

Zadnje što je potrebno je modifikacija sučelja za korutinu i promise tipa. U ugnježđenoj klasi *promise_type* smo dodali atribut `back_value` preko kojeg spremamo vrijednost koja je predodređena za korutinu, a poslana iz pozivnog programa. Sljedeće što je različito je `yield_value()` metoda. Ona, osim što sprema vrijednost koja je poslana kao argument operatora `co_yield()`, čisti `back_value` vrijednost i vraća *BackAwaiter* objekt (dok se u primjeru 1.8 vraćao `std::suspend_always`). U sučelju za korutinu dodali smo `setBackValue()` metodu koja sprema vrijednost u varijablu promise objekta. Implementacija ostalih metoda je identična kao u 1.9

`co_return`

`co_return` je izraz koji označava trenutnu točku završetka korutine. Kada korutina dosegne `co_return`, ona se suspendira, a rezultat koji se specificira u izrazu vraća se pozivatelju korutine. Ovaj izraz omogućuje elegantno označavanje kraja korutine i prijenos rezultata nazad. Kako proširenje promise tipa i sučelja za korutinu je slično kao u prošlom poglavlju objasnit ćemo u kratkom primjeri koje su razlike.

```
1  #include <iostream>
```

```
#include <coroutine>
#include "task.h"

Task gauss_sum(int n){
    int sum = 0;
    for(int i = 1; i <= n; ++i){
        sum += i;
        co_await std::suspend_always{};
    }
    co_return sum;
}

int main(){
    auto gauss = gauss_sum(10);
    while(gauss.resume())
    {
        std::cout<<"Nastavi"<<std::endl;
    }

    std::cout<<"Krajnji rezultat je: " << gauss.getResult()<<std::endl;
    return 0;
}
```

Listing 1.15: Simulacija Gaussove sume

U primjeru 1.15 vidimo jednostavnu simulaciju Gaussove dosjetke. Metode sučelja za korutine imaju iste zadaće kao u ovom primjeru samo što sada `getResult()` vraća vrijednost od `co_return`. Pogledajmo kakve promjene ima *Promise* klasa kada se upotrebljava `co_return`.

```
struct Promise{
    int final_result;
    ...
    void return_value(const auto& val)
    {
        final_result = val;
    }
};
```

Listing 1.16: *Promise* klasa

Ključna stvar je implementirati funkciju `return_value()` koja se poziva nakon pojave ključne riječi `co_return` koja prima neku vrijednost kao argument. U primjeru je jednostavna implementacija kod koje samo tu vrijednost sprema u atribut `final_result`. Primjetimo da se taj atribut mijenja samo jednom za razliku od slučaja kod `co_yield` kod kojeg se mijenjanje događa kod svake suspenzije. Ako ne prosljeđuje nikakav argument, odnosno imamo `co_return` koji ne vraća nikakvu vrijednost, trebamo implementirati metodu `return_void()`.

```
#include <coroutine>
#include "promise.h"

class [[nodiscard]] Task{
private:
    std::coroutine_handle<promise_type> cHandle;
public:
    ...
    int getResult(){
        return cHandle.promise().final_result;
    }
};
```

Listing 1.17: Sučelje za korutinu

Kod klase *Task* se pojavljuje nova metoda `getResult()` koja ima jednaku zadaću kao i `getValue()` u 1.9, dohvaća vrijednost iz promise tipa i vraća ga u pozivni program.

Napomenimo da trebamo paziti na nepoželjno ponašanje kod `co_return` izraza. Naime, nedefiniran slučaj je kada želimo koristiti `co_return` i kao izraz s kojim vraćamo vrijednost i kao izraz s kojim ne vraćamo vrijednost u istom primjeru. Dakle, klasa *Promise* ne smije imati obadvije funkcije (`return_void() i return_value()`) implementirane.

1.4 Operator `co_await`, Awaitable i Awaiter

Već smo spomenuli da je `co_await` unarni operator koji se pojavljuje samo u kontekstu korutina. Nadalje, znamo da tip koji taj operator podržava je *Awaitable* tip. Cilj ovog poglavlja je prikazati kako kompajler pristupa tom operatoru kada susretne tip koji mu nije poznat. Do sada smo vidjeli primjere gdje `co_await` prima `Awaiter` iz standardne biblioteke ili gdje prima kao argument `Awaiter` implementiran od strane programera. Cilj ovog poglavlja je pokazati kako modificirati dodatne klase da bi `co_await` podržavao različite tipove kao argument funkcije. Ideja je da operator prima bilo koji tip te ga onda mapiramo u neki `Awaiter`.
U C++ postoje dva načina:

- `await_transform` metoda
- operator `co_await()`.

Metoda `await_transform` se pojavljuje kao funkcija članica odgovarajućeg promise tipa. Kod operatora `co_await()` je situacija malo drugačija jer njega bi trebalo implementirati za svaki tip za kojeg želimo da to funkcionira.

Metoda `await_transform` ima veći prioritet nego `co_await()` što znači da kod pojave `co_await` operatora s nekim izrazom kompajler prvo provjerava da li postoji ta metoda većeg prioriteta. Ako metoda ne postoji tada izraz koristimo kao izrazni *Awaitable*

objekt[2]. Nakon toga ide provjera o postojanju `co_await()` kod tog objekta. Ako ni on ne postoji tada se opet sam izraz koristi kao *Awaitable* objekt.

Pogledajmo kako možemo implementirati `await_transform` metodu.

```
struct Promise{
    ...
    auto await_transform(int value) {
        return MojImplementiranAwaiter{value};
    }
    ...
};
```

Listing 1.18: Implementacija `await_transform` metode

Primjer 1.18 prikazuje jedan način kako riješiti slučaj kada `co_await` primi `integer` kao tip. Vrijednost je proslijeđena u neki interni *Awaiter* koji ima implementiranu svoju logiku te omogućava da kompajler omogući ovakav rad. Nešto slično možemo napraviti za svaki drugi tip.

```
struct Promise{
    int current_value = 0;
    ...
    auto await_transform(int value) {
        current_value = value;
        return std::suspend_always;
    }
    ...
};
```

Listing 1.19: Zamjena za `co_yield`

Uz pomoć ovog pristupa možemo napraviti da `co_await` funkcionira na identičan način kao `co_yield`. U primjeru 1.19 vidimo da nakon što `co_await` primi vrijednost, sprema tu vrijednost u `current_value` varijablu te vraća standardni *Awaiter*. Ako se vratimo u 1.8 možemo primjetiti da `yield_value()` metoda ima istu implementaciju kao ova naša te na ovaj način zamjenjujemo `co_yield` operator s `co_await`.

Poglavlje 2

Biblioteka cppcoro

cppcoro je open-source C++ biblioteka koja pruža bogat skup alata za rad s korutinama i asinkronim programiranjem. Inicijalno je stvorena od Lewisa Bakera koji se posebno istaknuo u području C++ programiranja i asinkronog programiranja. cppcoro dodaje dodatne značajke i apstrakcije kako bi se olakšala izgradnja kompleksnih asinkronih sustava. Ova biblioteka pruža viši sloj apstrakcije koji omogućuje programerima rad s korutinama na visokoj razini bez gubitka kontrole nad niskim detaljima implementacije. Biblioteku se može naći kao git repozitorij stvoren od strane autora no taj repozitorij se ne održava već duže vrijeme. Umjesto tog repozitorija koristit ćemo repozitorij koji je redovno održavan i unaprijeđen sa novim karakteristikama u odnosu na stari(vidi [3]). Također se može očekivati daljnji razvoj te biblioteke.

Iz prošlog poglavlja smo mogli zaključiti da je sučelje za korutinu i pripadni promise tip obavezan za funkcioniranje korutina. Kroz biblioteku je to podržano kroz niz tipova za korutinu gdje je svaki promise tip ugniježden u svoj tip korutine. U biblioteci također možemo naići na niz *Awaitable* tipova i niz funkcija koji ovise o *Awaitable* tipovima i javljaju se uz co_await operator. Osim toga biblioteka podržava princip umrežavanja, odnosno ima socket klase za primanje i slanje podataka preko mreže. U ovom trenutku je podržano samo na Windows platformi, a za Linux se očekuje da će biti podržavan uskoro. Trenutno podržava samo TCP/IP, UDP/IP preko IPv4 i IPv6 protokola [3]. Uz to se još javljaju alati za upravljanje s I/O algoritmima i shemama te alati za manipulaciju datoteka. Za kraj treba spomenuti da se u biblioteci nalaze određene metafunkcije i koncepti koji omogućuju korisniku za još bolje upravljanje korutinama.

U ovom poglavlju nećemo ulaziti u srž implementacije cppcoro biblioteke i analizirati kako je nešto napravljeno. Cilj poglavlja je prikazati kako se biblioteka koristi te koja je svrha pojedinih korutinskih tipova, *Awaitable* tipova i ostalih nekih stvari koje se pojavljuju u cppcoro biblioteci.

2.1 Tipovi korutine

Dok govorimo o tipovima korutine u `cppcoro` biblioteci mislimo na sučelje za korutinu i promise tip zajedno. Tipovi korutine na koje možemo naići su sljedeći:

- `task<T>`
- `shared_task<T>`
- `generator<T>`
- `recursive_generator<T>`
- `async_generator<T>`.

Sve klase su implementirane kao template klase kako bi se mogle koristiti za sve tipovi. Krenimo sada s objašnjenjem klasa. Klasa `task<T>` ima dosta sličnosti kao i klasa iz prvog poglavlja. Izvršava se lijeno, odnosno ne počne se izvršavati kod inicijalizacije nego prvi put nakon što se pozove od strane neke korutine. Može koristiti samo `co_await` i `co_return` (`co_yield` je namijenjen za generatore).

```
#include <iostream>
#include <cppcoro/task.hpp>
#include <cppcoro/sync_wait.hpp>

cppcoro::task<int> druga_korutina(){
    std::cout << "druga: vracam vrijednost" << std::endl;
    co_return 1;
}

cppcoro::task<> prva_korutina() {
    std::cout << "prva: ulazak" << std::endl;
    auto druga = druga_korutina();
    auto vrijednost = co_await druga;
    std::cout << "prva: izlazak - " << vrijednost << std::endl;
}

cppcoro::task<> primjer(){
    cppcoro::task<> pk_task = prva_korutina();
    std::cout << "primjer: Zavrsila inicijalizacija" << std::endl;
    co_await pk_task;
    std::cout << "primjer: Kraj korutine" << std::endl;
}

int main() {
    std::cout<< "main: Pocetak" << std::endl;
    cppcoro::sync_wait(primjer());
```

```
27      std::cout << "main: Kraj" << std::endl;
28      return 0;
29  }
```

Listing 2.1: Primjena `cppcoro::task`

Proučimo sada jednostavan primjer 2.1. U glavnom dijelu programa primjećujemo metodu `sync_wait()` koja služi samo da neki *Awaitable* završi do kraja. Ta funkcija se javlja zato što u glavnom dijelu programa ne smijemo koristiti operatore za korutine jer bi onda sam glavni dio bio korutina. Korutina `primjer()` prvo inicijalizira drugu korutinu, a nakon toga poziva `co_await` operator na `task` objektu od te druge korutine. To nam služi kako bi program čekao izvršenje pozvane korutine, a tek onda nastavio izvršavanje korutine `primjer()`. Kod korutine `prva_korutina()` je sve isto osim što ona dobiva neku povratnu vrijednost od `druga_korutina()` jer je ona `task` klasa koja vraća `int` vrijednost. Ispis koji možemo očekivati je sljedeći:

```
main:  Pocetak
primjer:  Zavrsila inicijalizacija
prva:  ulazak
druga:  vracam vrijednost
prva:  izlazak - 1
primjer:  Kraj korutine
main:  Kraj
```

Klasa `shared_task<T>` je sličan tip korutine kao `task<T>`. Ima isto izvršavanje i vraća jednu vrijednost asinkrono. Jedina razlika je u tome što `shared_task<T>` vrijednost može dopuštati mnogo referenci na kreiranu vrijednost od korutinskog tipa te se može kopirati ta vrijednost. Također može više korutina čekati završetak `shared_task<T>` objekta da nastave svoje izvršenje.

Što se tiče generatora, trenutno imamo 3 različita u biblioteci. Prvi, `generator<T>`, je sličan našem generatoru iz 1.11 primjera. Korutina vraća vrijednosti tipa `T` kroz `co_yield` operator kao u spomenutom primjeru te ima implementiran iterator u sebi sa potrebnim metodama: `begin()`, `end()`, `operator++()` i `operator*()`. Izvršavanje tog generatora je također lijeno. Pogledajmo sada primjer s nizom Fibonaccijevih brojeva iz 1.10. Potrebne modifikacije da kod proradi s `cppcoro` bibliotekom su minimalne: zamijeniti header datoteku za generator s ovim iz biblioteke

`#include<cppcoro/generator.hpp>`,

zamijeniti tip iz primjera s tipom

`cppcoro::generator<int>`

i ispis će biti identičan kao u tom primjeru. Ovaj generator nema metodu `getValue()` nego elementima pristupa samo preko iteratora s pripadajućim operatorom (`operator*()`). Kao što u task klasama ne smijemo koristiti `co_yield` tako u slučaju generatora ne možemo koristiti `co_await`.

Drugi generator je `recursive_generator<T>`. On preko `co_yield` operatora, osim što može vratiti vrijednost tipa `T`, može vratiti vrijednost tipa `recursive_generator<T>` i u tom slučaju su svi elementi novog vraćenog rekurzivnog generatora također elementi početnog generatora. Kada iteriramo kroz elemente generatora tada `operator++` od iteratora klase točno zna koja je najdublja korutina koja daje sljedeću vrijednost. Pogledajmo primjer Fibonaccijevih brojeva s rekurzivnim generatorom.

```
#include <iostream>
#include <cppcoro/recursive_generator.hpp>

cppcoro::recursive_generator<int> fibonacci(int a, int b, int depth)
{
    co_yield a;

    if (depth > 0)
    {
        co_yield fibonacci(b, a + b, depth - 1);
    }
}

int main()
{
    const int depth = 10;

    for (auto value : fibonacci(0, 1, depth))
    {
        std::cout << value << " ";
    }

    return 0;
}
```

Listing 2.2: Rekurzivni generator

U ovoj klasi također ne možemo koristiti ključnu riječ `co_await`.

Zadnji generator je `async_generator<T>` koji može stvarati vrijednosti asinkrono, što je razlika od ostalih generatora koje smo spomenuli. Nema ni restrikcija za korištenje operatora dakle dopušteni su i `co_await` i `co_yield`. Korištenje ovog generatora je slično ostalima tako da nećemo ulaziti u detalje. Za više informacija možete pogledati u službenoj

dokumentaciji biblioteke.

2.2 *Awaitable* tipovi

Biblioteka `cppcoro` je dobro potkrijepljena i *Awaitable* tipovima. Trenutno ih ima devetero i služe za jako delikatnu manipulaciju korutinama i njihovog izvršavanja. Tipovi koji su dio biblioteke su sljedeći:

- `single_consumer_event`
- `single_consumer_async_auto_reset_event`
- `async_mutex`
- `async_manual_reset_event`
- `async_auto_reset_event`
- `async_latch`
- `sequence_barrier`
- `multi_producer_sequencer`
- `single_producer_sequencer`.

U ovom potpoglavlju ćemo ukratko objasniti u kojim situacijama koristimo koji tip, a za neke će biti prikazan primjer koda za bolje razumijevanje.

Tip `single_consumer_event` je vrlo jednostavan tip događaja s resetiranjem i podržava samo jednu korutinu. Za bolje razumijevanje pogledajmo primjer.

```
1  #include <cppcoro/single_consumer_event.hpp>
2  #include <cppcoro/task.hpp>
3  #include <cppcoro/sync_wait.hpp>
4
5  #include <chrono>
6  #include <iostream>
7  #include <thread>
8  #include <future>
9
10 cppcoro::single_consumer_event event;
11
12 cppcoro::task<> korutina()
13 {
14     co_await event;
```

```
15
16      std::cout << "Dobio sam dopustenje" << std::endl;
17  }
18
19  void izvrsi()
20  {
21
22      std::cout<<"Cekam 3 sekunde"<<std::endl;
23      using namespace std::chrono_literals;
24      std::this_thread::sleep_for(3s);
25      event.set();
26  }
27
28  int main() {
29      auto con = std::async([]{ cppcoro::sync_wait(korutina()); });
30      auto prod = std::async(izvrsi);
31
32      con.get(), prod.get();
33
34      return 0;
35  }
```

Listing 2.3: Tip `single_consumer_event`

Vidimo da se u korutini javlja objekt tipa `single_consumer_event` kao argument operatora `co_await`. U tom trenutku korutina se suspendira i nastavit će se kada neka druga dretva pozove metodu `set()` na objektu `event`. U glavnom dijelu programa koristimo `std::async()` koja pokreće dvije nove dretve. `std::async` je funkcija u C++ standardnoj biblioteci <`future`> koja omogućuje asinkrono izvođenje funkcija. Ova funkcija se koristi za pokretanje funkcije u zasebnoj dretvi (eng. *thread*) i vraća `std::future` objekt koji omogućuje pristup rezultatu izvršavanja funkcije. Ispis programa je sljedeći:

```
Cekam 3 sekunde
Dobio sam dopustenje
```

Iako dretva za korutinu starta prva, ona se suspendira i čeka da druga dretva izvrši čekanje i pozove potrebnu metodu. Korutina u slučaju ovog tipa može biti suspendirana samo jednom. Dakle, za svaki sljedeći poziv `co_await event;` korutina bi nastavila svoje izvršavanje i ne bi čekala da neka druga dretva pozove metodu `set()` na odgovarajućem objektu.

Kako bi riješili taj problem, odnosno omogućili korutini da se suspendira više puta s *Awaitable* tipom i nastavili pozivom metode `set()` iz neke druge dretve koristimo sličan tip `single_consumer_async_auto_reset_event`. Korištenje tog tipa je identično kao kod `single_consumer_event` tipa samo što ima to vrlo korisno dodatno svojstvo. Ovaj

tip može biti koristan u slučaju kada korutina obavlja zadaću svaki put kad se javi promjena nekog parametra. Ponovimo da ovdje također može čekati samo jedna dretva kao i kod prvog tipa. Ako želimo omogućiti da više dretvi čeka u trenutku suspenzije i čeka da ih neka druga dretva nastavi koristit ćemo `async_auto_reset_event`. Ovaj tip se isto koristi kao prošla dva.

Možemo zaključiti da sve što može tip `single_consumer_async_auto_reset_event`, može također i `async_auto_reset_event`. Isti odnos imaju drugi i prvi tip. Zašto onda uopće postoje prva dva tipa kada sve to može `async_auto_reset_event`? Kao što obično biva u programiranju, povećanje broja funkcionalnosti dovodi do smanjenja efikasnosti i brzine, što predstavlja problem i u ovom slučaju.

Tip koji je sličnog imena kao `async_auto_reset_event` i ima sličnu fukcionalnost kao on zove se `async_manual_reset_event`. Već po imenu se može zaključiti da će se neke stvari mijenjati ručno od strane programera. Navedimo prvo koje su sličnosti između ova dva tipa. Tip `async_manual_reset_event` također dopušta jednu ili više dretvi da čeka poziv metode `set()` za nastavak izvršavanja. Ima dva različita stanja: `'set'` i `'no set'` gdje kod prvog stanja se dretva ne suspendira dok je kod drugog stanja dretva suspendirana kada naiđe na `co_await` s tim tipom (također sve isto kao s prošlim tipom). Početno stanje kod inicijalizacije oba tipa je stanje `'no set'`. Objasnimo sada razliku. Kod tipa `async_auto_reset_event` kada neka dretva biva suspendirana ona čeka metodu `'set'` od neke druge dretve da nastavi svoje izvršavanje. Nakon što druga dretva pozove metodu `set()` na objektu tog tipa, korutina nastavi svoje izvršavanje, ali stanje u objektu tipa `async_auto_reset_event` se automatski mijenja u `'no set'`. Zbog toga će ostale korutine koje dođu do točke suspenzije trebati ponovno čekati poziv metode `set()` za svoj nastavak. Kod tipa `async_manual_reset_event` je situacija drugačija na način da se stanje objekta, nakon poziva metode `set()`, ne vraća u `'no set'` nego ostaje u stanju `'set'`. Zbog toga nova korutina koja naiđe na točku suspenzije ne bi bila suspendirana nego bi nastavila svoje izvršavanje. Da se to promjeni trebalo bi promjeniti stanje u objektu s metodom `reset()` koja vraća objekt u stanje `'no set'`.

Sljedeći tip je `async_mutex`. Koristi se u slučajevima gdje više dretvi pristupa istom dijeljenom podatku te mu je zadaća da spriječi istovremeno pristupanje podatku. Pogledajmo ponovljeni primjer Fibonaccijevog niza kako bi objasnili primjenu.

```
1  #include <cppcoro/sync_wait.hpp>
2  #include <cppcoro/task.hpp>
3  #include <cppcoro/async_mutex.hpp>
4  #include <iostream>
5  #include <vector>
6  #include <thread>
7
8
9  cppcoro::async_mutex mutex;
10
```

```
11 std::vector<int> fibonacciNumbers;
12
13 cppcoro::task<> generateFibonacci(int count) {
14     cppcoro::async_mutex_lock lockFibonacci = co_await mutex.
       scoped_lock_async();
15
16     int a = 0, b = 1;
17     for (int i = 0; i < count; ++i) {
18         fibonacciNumbers.push_back(a);
19         int next = a + b;
20         a = b;
21         b = next;
22     }
23 }
24
25 int main() {
26     std::vector<std::thread> threads(5);
27
28     for (auto& thread : threads) {
29         thread = std::thread([] {
30             cppcoro::sync_wait(generateFibonacci(10));
31         });
32     }
33
34     for (auto& thread : threads) {
35         thread.join();
36     }
37
38     for (int num : fibonacciNumbers) {
39         std::cout << num << " ";
40     }
41     std::cout << std::endl;
42
43     return 0;
44 }
```

Listing 2.4: Tip `async_mutex`

U glavnom programu stvaramo pet dretvi koje pokreću istu korutinu. Nakon što prva dođe do suspenzije (14. linija koda) zaključava ostatak korutine za ostale dretve. U tom trenutku su ostale dretve sve suspendirane osim one koja je zaključala ostatak korutine. Ona nastavlja svoje izvršavanje tako dugo dok ne završi do kraja, odnosno dok ne izgenerira niz od 10 Fibonaccijevih brojeva. Nakon toga se ostatak korutine otključa za sljedeću dretvu dok su ostale i dalje suspendirane. Postupak se ponavlja sve dok svaka korutina ne generira svoj niz od 10 Fibonaccijevih brojeva i doda ga u vektor. Taj vektor je podatak koji je dijeljen između svih dretvi i koji se štiti mutexom. Ispis je sljedeći:

0 1 1 2 3 5 8 13 21 34 0 1 1 2 3 5 8 13 21 34 0 1 1 2 3 5 8 13 21 34 0 1 1 2 3 5 8 13 21 34 0 1 1 2 3 5 8 13 21 34

Može se vidjeti da svaka dretva završi svoje generiranje prije nego što sljedeća počinje s istim. Možemo primjetiti da je `async_mutex` sinkronizacijski mehanizam jer izvršavanje ide slijedno, jedno iza drugog.

Situacija kod `async_latch` je puno jednostavnija. On dopušta korutinama da čekaju tako dugo dok se brojač objekta ne smanji do nule. Kada brojač stigne do nule tada je spreman i nastavlja sve korutine. Objekt tog tipa je nakon toga spreman i ne suspendira ni jednu korutinu tako dugo dok se objekt ne uništi.

```
#include <cppcoro/sync_wait.hpp>
#include <cppcoro/async_latch.hpp>
#include <cppcoro/task.hpp>
#include <chrono>
#include <iostream>
#include <future>

using namespace std::chrono_literals;


cppcoro::async_latch latch(3);

cppcoro::task<> odbrojavanje() {
    std::cout << "Prije tocke suspenzije" << std::endl;
    co_await latch;
    std::cout << "Nakon tocke suspenzije" << std::endl;
}

void smanjuj_counter(){

    for(int i = 0; i < 3; ++i){
        std::cout << "Dekrement"<< std::endl;
        latch.count_down();
        std::this_thread::sleep_for(1s);
    }

}

int main(){

    auto korutina = std::async([]{ cppcoro::sync_wait(odbrojavanje());
    });
    std::this_thread::sleep_for(1s);
    auto pom = std::async(smanjuj_counter);
```

```
34
35      korutina.get(), pom.get();
36
37      return 0;
38  }
```

Listing 2.5: Tip `async_latch`

U primjeru 2.5 se vidi kako to funkcionira. Imamo dvije dretve gdje je jedna korutina, a druga funkcija koja smanjuje brojač objekta `latch` svaku jednu sekundu. Pošto je korutina suspendirana do trenutka kada brojač nije jednak nuli ispis je sljedeći:

```
Prije tocke suspenzije
Dekrement
Dekrement
Dekrement
Poslije tocke suspenzije
```

Zadnja tri tipa nećemo objašnjavati detaljno. Objasnit ćemo za što služe i gdje se koriste, a za detalje pogledajte službenu dokumentaciju.

`sequence_barrier`, `single_producer_sequencer` i `multi_producer_sequencer` su ključni pojmovi u sustavima proizvođača i potrošača (eng. *producer-consumer systems*) te se koriste za efikasnu sinkronizaciju i upravljanje redoslijedom događaja ili poruka u višenitnim okruženjima.

`sequence_barrier` je alat koji omogućuje koordinaciju između više potrošača i jednog proizvođača. On prati redoslijed objavljenih sekvenci, omogućujući potrošačima da čekaju na određene sekvence prije nego što nastave s obradom. Ova vrsta alata ključna je za održavanje ispravnog redoslijeda događaja u višenitnim sustavima.

`single_producer_sequencer` je sekvencer koji podržava samo jednog proizvođača. Ovaj pristup pojednostavljuje implementaciju i optimizira performanse u situacijama kada postoji samo jedan izvor podataka. Koristi se za praćenje redoslijeda sekvenci koje generira jedan proizvođač, olakšavajući time upravljanje redoslijedom događaja.

`multi_producer_sequencer`, s druge strane, podržava više proizvođača. Ovaj sekvencer omogućuje više izvora podataka da objavljuju sekvence u zajednički redoslijed. To je korisno u sustavima gdje postoji više izvora informacija, omogućavajući im da koordinirano doprinose redoslijedu događaja ili poruka.

2.3 Pomoćne funkcije

Ovo poglavlje je namijenjeno za funkcije koje biblioteka pruža. Neke od njih smo već koristili za pokretanje programa dok će druge ovdje biti objašnjene. Funkcije u biblioteci

su sljedeće:

- `sync_wait()`
- `when_all()`
- `when_all_ready()`
- `fmap()`
- `schedule_on()`
- `resume_on()`.

Korištenje funckcije `sync_wait()` vidjeli smo u primjeru 2.2. Služi za pokretanje zadatka (`task`) i za sinkronizirano čekanje zadatka (`task`) većinom iz glavnog dijela programa.

Objasnimo sada `when_all()` funkciju. Funkcija se javlja kao argument operatora `co_await` i stvara nove *Awaitable* (odnosno nove korutine) koje se tada pokreću sinkronizirano, jedna iza druge. Broj novih korutina koje može stvoriti je proizvoljan i konačan. Svaka ta korutina može i ne mora vratiti vrijednost. Niz vrijednosti vraćene od svake korutine vraća se u `std::tuple` ili `std::vector`. Pogledajmo kroz primjer kako to funkcionira.

```
#include <cppcoro/task.hpp>
#include <cppcoro/when_all.hpp>
#include <cppcoro/sync_wait.hpp>
#include <iostream>
#include <future>
#include <string>

using namespace std::chrono_literals;

cppcoro::task<std::string> prvi_task(){
    std::this_thread::sleep_for(3s);
    std::cout << "Ispis: prvi task" << std::endl;
    co_return "Kraj";
}
cppcoro::task<int> drugi_task(){
    std::cout << "Ispis: drugi task" << std::endl;
    std::this_thread::sleep_for(2s);
    co_return 5;

}
cppcoro::task<> example()
{
```

```
23      auto [a, b] = co_await cppcoro::when_all(prvi_task(), drugi_task());
24
25      std::cout << "example: Zavrsili su - " << a << ", " << b << std::
        endl;
26  }
27
28  int main()
29  {
30      auto korutina = std::async([]{ cppcoro::sync_wait(example()); });
31
32      return 0;
33  }
```

Listing 2.6: Funkcija `when_all()`

Korištenje funkcije `when_all()` vidimo u korutini `example()`. Ona stvara dvije nove korutine te se izvršavaju jedna iza druge. Zbog toga je ispis ovog programa sljedeći:

```
Ispis:  prvi task
Ispis:  drugi task
example:  Zavrsili su - Kraj, 5
```

Zbog sinkronizacije se ispis iz drugog zadatka pojavljuje nakon prvog iako prvi zadatak ima stajanje od nekoliko sekundi. Također vidimo da su povratne vrijednosti vraćene kroz `std::tuple`.

Problem koji se javlja kod te metode je da nema dobru reakciju na neočekivano zaustavljanje korutine, odnosno ne izvršavanje te korutine do kraja. Naime, s tom funkcijom ne možemo utvrditi u kojoj je korutini problem, odnosno koja korutina se zaustavila. Da bi utvrdili tako nešto koristimo metodu `when_all_ready()`. Ona, osim što ima iste karakteristike kao prethodna metoda, daje informaciju u kojoj korutini je problem kod zaustavljanja. Tu informaciju dohvaćamo kroz metodu `result()` na vrijednost koju je korutina vratila (u primjeru 2.6 bi to bilo `a.result()`).

Funkcija `fmap()` se koristi za mapiranje vrijednosti unutar objekta koji podržava korutine (npr. `generator`) kroz funkciju transformacije (funktor) ili lambda izraz.. Ima dva argumenta funkcije gdje je prvi normalna funkcija koja će se izvršavati na svakoj vrijednosti iz objekta korutinskog tipa, a drugi je taj tip korutine. Tipovi korutine koji se najčešće koriste su: `generator<T>`, `recursive_generator<T>`, `async_generator<T>`, a za promjenu pojedinačnih vrijednosti možemo koristiti `task<T>` i `shared_task<T>` klase.

```
1  #include <cppcoro/fmap.hpp>
2  #include <cppcoro/generator.hpp>
3
4  int kvadriraj(int value);
5
```

```
6 cppcoro::generator<int> gen();
7
8 cppcoro::generator<int> zadatak = cppcoro::fmap(kvadriraj, gen());
```

Listing 2.7: Funkcija `fmap()`

Primjer 2.7 prikazuje skicu koda za metodu `fmap()`. Kao argumente prima metodu koja kvadrira član i generator koji ima cijele brojeve u sebi. Povratni tip funkcije je također generator istog tipa kao povratna vrijednost funkcije `kvadriraj`.

Za preostale dvije metode nećemo ulaziti u detalje, ali vrijedi napomenuti da je sintaksa korištenja slična kao kod ostalih metoda.

2.4 Ostatak biblioteke

U ovom poglavlju ćemo se posvetiti ostatku biblioteke te ukratko objasniti koje još alate biblioteka posjeduje. Govorit ćemo o tome kako se suočava s otkazivanjem i planiranjem događaja u asinkronom programiranju, kakve alate posjeduje za rad kod umrežavanja (eng. *networking*) te ostalo.

Što se tiče alata za otkazivanje, biblioteka ima sljedeće implementirane klase:

- `cancellation_token`
- `cancellation_source`
- `cancellation_registration`

. Klase su međusobno povezane te ih koristimo istovremeno sve zajedno. Prvi od njih, `cancellation_token`, je vrijednost koja se može proslijediti nekoj funkciji te služi za prekidanje izvršavanja funkcije od strane pozivatelja. Objekt klase `cancellation_token` se može kreirati samo preko `cancellation_source` objekta te preko njega pozivatelj funkcije manipulira tokenom usred izvršavanja. `cancellation_registration`, koristimo kako bi odredili ponašanje programa u slučaju otkazivanje neke funkcije preko tokena.

Što se tiče planiranja izvršavanja dretvi koristimo `static_thread_pool`. Ono nam dopušta da na skupu dretvi (gdje je broj dretvi fiksiran) kvalitetno raspoređujemo posao.

Krenimo sada na umrežavanje. Najbitnija klasa za umrežavanje u ovoj biblioteci je `socket` te služi za asinkrono primanje i slanje podataka. Trenutno je broj paradigmi koje podržava dosta malen, a među njima su TCP/IP i UDP/IP paradigma. Uz klasu `socket` implementirane su klase koji služe za reprezentaciju IP adrese ili reprezentaciju IP adrese i porta zajedno. Napravljene su ovisno o kojem internet protokolu je riječ (IPv4 ili IPv6). Riječ je o sljedećim klasama:

- `ip_address`
- `ip_endpoints`
- `ipv4_address`
- `ipv4_endpoint`
- `ipv6_address`
- `ipv6_endpoint`.

Klase koje u sebi `address` klase služe za reprezentaciju IP adrese dok `endpoints` klase, osim za reprezentaciju IP adrese, služi i za reprezentaciju porta. Ako klasa u svojem nazivu nema oznaku verzije protokola to znači da može reprezentirati bilo koji od njih.

Metafunkcije koje se javljaju su `awaitable_traits<T>` i `is_awaitable<T>`. Prva služi za određivanje tipa koji će nastat nakon `co_await` izraza, a druga nam govori može li na dani tip biti pozvan `co_await` operator. Osim metafunkcija, neki koncepti su isto implementirani: `Scheduler` i `DelayedScheduler` (služe za planiranje određenog rad dretvi), `Awaitable<T>` i `Awaiter<T>` (služe za otkrivanje tipa koji se može koristiti kao *Awaitable* ili *Awaiter*.

Poglavlje 3

Korutine u drugim programskim jezicima

3.1 Python

U Pythonu, korutine su poseban oblik funkcija koje omogućuju prekidanje izvršavanja kako bi se izvršili drugi zadaci, čime se postiže konkurentnost i efikasnost. Možemo zaključiti da imaju identičnu ulogu kao u C++. Za implementaciju korutina koristimo ključne riječi `async` i `await`. Za razliku od programskog jezika C++, nije potreban nikakav dodatan posao programera za korištenje korutina.

```python
import asyncio

async def some_async_function():
    return 20


async def example_coroutine():
    print("Start coroutine")
    result = await some_async_function()
    print(f"Coroutine finished with result: {result}")


def main():
    asyncio.run(example_coroutine())

if __name__ == "__main__":
    main()
```

Listing 3.1: Primjer korutine u Pythonu

Pogledajmo sada primjer 3.1 koji nam pokazuje ključne stvari kod definiranja korutina. Uz definiciju metoda nailazimo na ključnu riječ `async`. Ona se koristi pri definiciji korutine, tj. svaka funkcija označena s `async def` smatra se korutinom. Ključna riječ `await` koristi se unutar korutine kako bi se čekao rezultat drugih korutina ili asinkronih operacija. `await` se koristi za označavanje mjesta gdje se izvršavanje može privremeno zaustaviti dok se ne dovrši određena asinkrona operacija. Ispis primjera bio bi sljedeći:

```
Start coroutine
Coroutine finished with result:  20
```

Kad se priča o korutinama u pythonu treba spomenuti modul `asyncio`. Naime, on pruža cijelu infrastrukturu za pisanje asinkronog koda i upravljanje korutinama. Koristi se za planiranje izvršavanja korutina, čekanje na završetak asinkronih operacija i slične zadatke. Modul `asyncio` u Pythonu koristi se ne samo za implementaciju korutina, već i za širok spektar asinkronih operacija i zadatka koji se mogu odnositi na različite aspekte programiranja, posebice u kontekstu mrežnog programiranja, web programiranja, I/O operacija, bazama podataka i mnogim drugim. U ovom potpoglavlju samo ćemo ukratko objasniti spomenuti modul u kontekstu korutina.

U primjeru koji smo prethodno prikazali, koristili smo jednu od metoda iz te biblioteke. Radi se o metodi `run()` te služi za pokretanje korutine. Nakon toga treba spomenuti metodu `create_task()` koja služi za pakiranje korutine u jedan zadatak koji će se naknadno izvršiti. U primjeru 3.2 vidimo kako bi uz ovu metodu zamijenili `main` funkciju iz prošlog primjera

```
async def main():
    task1 = asyncio.create_task(example_coroutine())

    await task1
```

Listing 3.2: Metoda `create_task()`

Uz metodu `sleep()` zaustavljamo dretvu na određen broj sekundi koji zadamo preko argumenta funkcije dok funkcija `gather()` služi za pokretanje više zadataka istovremeno.

```
import asyncio

async def async_task1():
    print("Task 1 started")
    await asyncio.sleep(1)
    print("Task 1 completed")

async def async_task2():
    print("Task 2 started")
    await asyncio.sleep(2)
    print("Task 2 completed")
```

```
12
13 async def main():
14     await asyncio.gather(
15         async_task1(),
16         async_task2()
17     )
18
19 if __name__ == "__main__":
20     asyncio.run(main())
```

Listing 3.3: Metoda `gather()` i `sleep()`

Primjer 3.3 prikazuje istovremeno pokretanje dva zadatka (metoda `gather()` može primiti više od dvije korutine kroz argument funkcije) te zaustavljanje svaku od dretvi na nekoliko sekundi. Kroz ispis ovog programa možemo zaista primjetiti da se zadaci pokreću istovremeno jer se ispis o startanju drugog zadatka javlja prije nego ispis o završetku prvog.

Biblioteka također ima metodu `shield(task)` koja štiti task od prekida u slučaju da neka druga dretva pozove taj prekid (moguće uz `task.cancel()`).

Nužne stvari u asinkronom programiranju je upravljanje vremenom izvršavanja asinkronih zadataka i korutina. Za takvo nešto biblioteka pruža podršku kroz sljedeće funkcije:

- `timeout`
- `timeout_at`
- `wait_for`
- `wait`

.

Metoda `timeout` služi za postavljanje globalnog vremenskog ograničenja na čitavu petlju događaja. U slučaju da neki zadatak ne završi unutar određenog vremena, doći će do iznimke `asyncio.TimeoutError`.

```
1 async with asyncio.timeout(5):
2     await neka_korutina()
```

Listing 3.4: Metoda `timeout`

Kod `timeout_at` je situacija slična. Ona postavlja vremensko ograničenje na određeni datum i vrijeme (primjer 3.5). Korisnost ove funkcije se primjećuje kada postavljamo apsolutno vremensko ograničenje

```
1 end_time = loop.time() + 5
2 async with asyncio.timeout_at(end_time):
```

```
3     await neka_korutina()
```

Listing 3.5: Metoda `timeout_at`

Metoda `wait_for` koristi se za postavljanje vremenskog ograničenja na pojedinačni asinkroni zadatak ili korutinu. U slučaju da se zadatak ne izvrši unutar određenog vremena javlja se iznimka `asyncio.TimeoutError`.

```
try:
    result = await asyncio.wait_for(neka_korutina(), timeout=3)
except asyncio.TimeoutError:
    print("Timeout occurred.")
```

Listing 3.6: Metoda `wait_for`

Metoda `wait` koristi se za istovremeno izvršavanje više asinkronih zadataka kao i funkcija `gather` samo što ovdje možemo staviti neko vremensko ograničenje.

```
tasks = [coroutine1(), coroutine2(), coroutine3()]
done, pending = await asyncio.wait(tasks, timeout=5, return_when=asyncio
    .ALL_COMPLETED)
```

Listing 3.7: Metoda `wait`

S ovim primjerom ćemo završiti priču o biblioteci `asyncio`. Vrijedi napomenuti da smo u ovom potpoglavlju samo malo zagrebli po površini spomenute biblioteke te da iza nje stoji još puno korisnih alata koji nisu pokazani.

Što se tiče generatora u Pythonu, oni se javljaju na drugačiji način nego u C++u. Ne dolaze u sklopu korutina nego se mogu nezavisno implementirati bez upotrebe ključne riječi `async`.

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

gen = fibonacci_generator()

for _ in range(10):
    print(next(gen))
```

Listing 3.8: Generator Fibbonaccijevih brojeva

Primjer 3.8 nam pokazuje generator Fibbonaccijevih brojeva, ovaj put u Pythonu. Koristi se ključna riječ `yield` (slično kao kod korutina u C++) te se definira bez potrebne ključne riječi za korutine.

3.2 Javascript

Programski jezik Javascript također ima bolju podršku za korutine nego C++ te nije potrebno raditi dodatne implementacije da cijela stvar funkcionira. Uz osnovne ključne riječi kojim definiramo korutine, JavaScript pruža podršku za korutine kroz razne biblioteke. U ovom potpoglavlju ćemo prikazati definiranje korutina te korištenje nekih bilioteka u svrhu dobre manipulacije s njima.

Ključne riječi za korištenje korutine su `async` i `await`. Riječ `async` se javlja kod definiranja funkcije i samim time označava funkciju korutinom. `await` se koristi u kontekstu korutina za čekanje na završetak izvršenja asinkronih operacija. Kada se koristi `await` ispred izraza koji vraća `Promise`, funkcija u kojoj se nalazi `await` će privremeno pauzirati izvršavanje dok se taj `Promise` ne ispunjava ili odbaci. Nakon što se `Promise` ispunjava, `await` vraća vrijednost na koju je `Promise` riješen. `Promise` u Javascriptu ima sličnu zadaću kao i u ostalim programskim jezicima. On je objekt koji predstavlja asinkroni zadatak koji će se izvršiti u budućnosti i koji može biti uspješno riješen, odbijen ili u stanju iščekivanja. Koristi se za upravljanje asinkronim operacijama i omogućava lakše rukovanje povratnim informacijama ili rezultatima tih operacija.

```
function fetchData() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('Data');
      }, 1000);
    });
  }

  async function myCoroutine() {
    console.log('Start coroutine');
    try {
      const data = await fetchData();
      console.log('Coroutine finished with data:', data);
    } catch (error) {
      console.error('Coroutine error:', error);
    }
  }

  myCoroutine();
```

Listing 3.9: Korutina u Javascriptu

Primjer 3.9 prikazuje jednostavnu korutinu `myCoroutine()` koja čeka na dohvaćanje podatka preko `await` od `Promise` objekta koji ima zadani neki timeout.

U ovom potpoglavlju ćemo spomenuti i ukratko objasniti dvije biblioteke `Bluebird` i `co`. `Bluebird` pruža mnoge mogućnosti u radu s `Promise` objektima te na taj način pruža kontrolu nad korutinama. `co` je starija biblioteka te se više ne koristi u tolikom obujmu, ali

vrijedi ju spomenuti.

Krenimo s `Bluebird` bibliotekom te pokažimo neke osnovne. Jedna od njih je funkcija `all`. Ona prima niz `Promise` objekta i vraća novi `Promise` koji se rješava kada svi `Promise` objekti u nizu završe, ili se odbacuje ako bilo koji od njih bude odbačeno.

```
const Promise = require('bluebird');

async function fetchData(value) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(value);
    }, Math.random() * 1000);
  });
}

async function fetchDataParallel() {
  const promise1 = fetchData('Value 1');
  const promise2 = fetchData('Value 2');

  try {
    const values = await Promise.all([promise1, promise2]);
    console.log(values); // ['Value 1', 'Value 2']
  } catch (error) {
    console.error(error);
  }
}

fetchDataParallel();
```

Listing 3.10: Metoda `all`

Ispis gornjeg objekta bi bio:

```
['Value 1', 'Value 2']
```

zato što će `all` pričekati sve korutine da završe i spremiti rezultate.

Slična metoda onoj prošloj je metoda `race`. Metoda `race` također prima niz `Promise` objekta i vraća novi `Promise` koji se rješava kada bilo koje od obećanja u nizu završi, s vrijednošću ili greškom tog obećanja. Ako u 3.11 zamijenimo `all` s metodom `race` tada bi u varijabli `values` dobili samo 'pobjednika' odnosno onu vrijednost kojoj je korutina brža.

```
const Promise = require('bluebird');

async function processData(data) {
  return new Promise(resolve => {
```

```
    setTimeout(() => {
      resolve(data.toUpperCase()); // Simulacija obrade podataka
    }, Math.random() * 1000);
  });
}

async function processDataList(dataList) {
  try {
    const results = await Promise.map(dataList, async data => {
      return await processData(data);
    });
    console.log('Processed data:', results);
  } catch (error) {
    console.error('Error:', error);
  }
}

const dataList = ['konj', 'lovac', 'dama', 'top', 'kralj'];

processDataList(dataList);
```

Listing 3.11: Metoda `map`

U ovom primjeru, funkcija `processData` simulira obradu podataka asinkronom operacijom s odgodom. Funkcija `processDataList` koristi `Promise.map` kako bi paralelno obradila svaki element iz liste podataka `dataList`. Svaki element se prosljeđuje funkciji `processData` koja vraća `Promise` s obrađenim podacima. Kada su svi podaci obrađeni, rezultati se ispisuju u konzolu. Na sličnu funkciju smo naišli kod `cppcoro` biblioteke s funkcijom `fmap`.

Jednaku zadaću ima i metoda `each` samo što ona iterira kroz elemente i modificira ih jednog po jednog, a ne paralelno.

Kako se ovdje radi o asinkronim operacijama, biblioteka također ima funkciju `delay` koja stopira izvršavanje dretve na određeno vrijeme što se uistino često javlja kod asinkronog programiranja.

Osim spomentih metoda postoji još veliki broj metoda u biblioteci (`any`, `props`, `join`...) s kojima se može manipulirati korutinama no nećemo ulaziti u detalje za svaku od njih.

Krenimo sada na generator te prikažimo na koji način se javljaju u JavaScriptu za razliku od C++. Generatore u Javascriptu implementiramo preko generatora funkcija. Generator funkcija je poseban tip funkcije koji omogućuje stvaranje generatora, koji je objekt s metodama za kontrolirano iteriranje kroz niz vrijednosti. Generator funkcija se definira s ključnom riječju `function*`, a koristi se `yield` izraz za pauziranje izvršavanja funkcije i vraćanje vrijednosti generatoru. Pogledajmo opet u primjeru 3.12 kako bi izgledao generator Fibbonaccijevih brojeva u JavaScriptu.

```
function* fibonacciGenerator() {
```

```
    let prev = 0;
    let curr = 1;

    while (true) {
      yield curr;
      [prev, curr] = [curr, prev + curr];
    }
  }

  const generator = fibonacciGenerator();

  for (let i = 0; i < 10; i++) {
    console.log(generator.next().value);
  }
```

Listing 3.12: Generator Fibbonaccijevih brojeva

Napomenimo sada neke razlike između `yield` u Javascriptu i `co_yield` u C++. Izraz `co_yield` se može koristiti za paralelno izvršavanje korutina dok to u ovom programskom jeziku nije omogućemo s `yield`. Naime, moguće je samo linearno izvršavanje što znači da jedan `yield` izraz čeka na završetak prije nego što se izvrši sljedeći. U C++ se `co_yield` može koristiti za očuvanja stanja između poziva te manipulirati njihova daljnja izvršavanja ovisno na stanje. Kod JavaScripta je situacija malo drugačija jer ne mogu lako održavati dugotrajna stanja između različitih poziva generatora.

Za kraj ovog potpoglavlja prokomentirat ćemo ukratko biblioteku `co`. Biblioteka `co` je bila popularna neko vrijeme za korištenje korutina u JavaScriptu. Ima niz sličnih/istih metoda kao `Bluebird` i imaju istu zadaću. Pojavom `async/await` sintakse smanjila se upotreba te biblioteke jer se na način `async/await` sintakse pojednostavilo korištenje asinkronih operacija. Biblioteka također ima manu jer ne rješava rukovanje greškama na najbolji način i lošije integrira korištenje `Promise` objekata.

3.3 Kotlin

U Kotlinu je situacija slična kao kod Pythona i JavaScripta. Ima jako dobru podršku za korutine te ne treba ništa dodatno implementirati. Uz ključne riječi za definiranje korutine, koristimo biblioteku `kotlinx.coroutines` koja sadrži bogate alate za asinkrono programiranje. Krenimo s jednostavnim primjerom da objasnimo ključne stvari.

```
import kotlinx.coroutines.*

suspend fun simulateAsyncProcessing(): Int {
    delay(1000) // Simulacija obrade koja traje 1 sekundu
    return 42
}
```

```
7
8  fun main() {
9      println("Pocetak programa")
10
11     val deferredResult = GlobalScope.async {
12         simulateAsyncProcessing()
13     }
14
15     runBlocking {
16         println("Cekam na rezultat...")
17         val result = deferredResult.await()
18         println("Dobijen rezultat: $result")
19     }
20
21     println("Kraj programa")
22 }
```

Listing 3.13: Primjer korutine u Kotlinu

Prva ključna riječ koju možemo primjetiti kod definicije funkcije je riječ `suspend`. Ona označava da se ta funkcija može pozivati iz drugih `suspend` funkcija ili korutina i može obavljati asinkronu obradu. Važan objekt za korutine koji se javlja u primjeru je `GlobalScope`. On je dio biblioteke `kotlinx.coroutines` i predstavlja glavni opseg korutina. Njegova funkcija članica `async{}` je tu kako bi pokrenula korutinu sa povratnom vrijednošću. Ako želimo korutinu bez povratne vrijednosti tada koristimo `launch{}`. U glavnom dijelu programa se nalazi i funkcija `runBlocking{}`. Ona se koristi za blokiranje glavne niti dok korutina ne završi i također pripada `kotlinx.coroutines` biblioteci. Za kraj spomenimo funkciju `await` koja ima funkciju da čeka završetak korutine koja je pokrenuta pomoću `async`. Ova funkcija, uz spomenuto čekanje na završetak asinkrone obrade, također vraća i rezultat u slučaju korutine s povratnom vrijednošću. Možemo ju koristiti samo unutar suspend funkcija ili korutina jer blokira izvršenje do dobijanja rezultata. Ako bismo pokušali koristiti `await` izvan suspend funkcije ili korutine, dobili bismo grešku kompilacije. U programu se koristi i funkcija `delay` koja je česta kod asinkronog programiranja i dio je spomenute biblioteke. Ispis programa je sljedeći:

```
Pocetak programa
Cekam na rezultat...
Dobijen rezultat:  42
Kraj programa
```

Za korutinu bez povratne vrijednosti (koje se stvaraju s `launch{}`) ne možemo koristiti `await` već funkciju `join`. Nju također treba koristiti unutar suspend funkcije ili korutine zbog istih razloga kao kod `await`.

U Kotlinu se također može korutina definirati lijeno. Pogledajmo na koji način.

```
import kotlinx.coroutines.*

suspend fun fetchData(id: Int): String {
    delay(1000)
    return "Podaci za id: $id"
}

fun main() = runBlocking {
    val deferred = async(start = CoroutineStart.LAZY) {
        fetchData(1)
    }

    deferred.start()
    println("Rezultat: ${deferred.await()}")

    println("Korutina je zavrsena")
}
```

Listing 3.14: Lijeno definiranje korutine

U ovom primjeru, koristimo opciju `start = CoroutineStart.LAZY` sa funkcijom `async` kako bismo odgodili pokretanje korutine. Zatim eksplicitno pokrećemo korutinu pozivom funkcije `start()` na objektu `deferred`. Nakon toga, koristimo funkciju `await()` kako bismo čekali na završetak korutine i dobili njen rezultat.

Krenimo sada s otkazivanjem (eng. *cancellation*) u Kotlinu. Za to se koriste funkcije `cancel()` i `cancelAndJoin()`. Prva funkcija otkazuje korutinu, dok druga funkcija, osim otkazivanja, čeka završetak korutine.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        repeat(10) { i ->
            println("Radim nesto korisno $i")
            delay(500)
        }
    }

    delay(2000) // Cekamo 2 sekunde

    println("Otkazujem korutinu")
    job.cancel() // Otkazujemo korutinu

    println("Cekamo na zavrsetak korutine")
    job.join() // Cekamo na zavrsetak korutine

```

```
19      println("Korutina je zavrsena")
20  }
```

Listing 3.15: Cancellation u Kotlinu

Primjer 3.15 prikazuje korištenje `cancel()`. Koristimo funkciju `launch` da bismo pokrenuli novu korutinu koja će ispisati poruke svakih 500 milisekundi. Nakon što prođe 2 sekunde, koristimo `cancel()` funkciju da bismo otkazali korutinu. Zatim koristimo `join()` funkciju kako bismo čekali da se korutina zaista završi prije nego što ispišemo završnu poruku. Ispis je sljedeći:

```
Radim nesto korisno 0
Radim nesto korisno 1
Radim nesto korisno 2
Radim nesto korisno 3
Otkazujem korutinu
Cekamo na završetak korutine
Korutina je zavrsena
```

Možemo primjetiti da se petlja ispisuje samo 4 puta zbog otkazivanja. Što se tiče funkcije `cancelAndJoin()`, ona radi istovremeno posao od dviju funkcija: `cancel()` i `join()`.

Za kraj, pokažimo nešto više o generatorima u Kotlinu. Pokazat ćemo dva načina kako se mogu stvoriti generatori: jedan uz Sequence API i drugi s korutinama.

Sequence API pruža mehanizam za lijeno generiranje elemenata sekvence. To omogućava generiranje elemenata sekvence po potrebi, što može biti efikasnije za velike sekvence. Sequence API koristi funkcije poput `generateSequence`, `sequence` i `yield` za definiranje generatora.

```
1  import kotlinx.coroutines.*
2
3  suspend fun fibonacciGenerator(): Sequence<Int> = sequence {
4      var prev = 0
5      var curr = 1
6      while (true) {
7          yield(curr)
8          val next = prev + curr
9          prev = curr
10         curr = next
11     }
12 }
13
14 fun main() = runBlocking {
15     val fibonacci = fibonacciGenerator()
16     println("Prvih 10 Fibonacci brojeva:")
```

```
    fibonacci.take(10).forEach { println(it) }
}
```

Listing 3.16: Generator uz pomoć Sequence API-ja

U primjeru 3.16, funkcija `fibonacciSequence` vraća sekvencu Fibonacci brojeva koristeći `sequence` funkciju iz Sequence API-ja. U tijelu sequence bloka, koristimo `yield` za generiranje Fibonaccijevih brojeva korak po korak.

Korutine u Kotlinu također mogu biti korištene za implementaciju generatora. Suspend funkcije u korutinama mogu pauzirati izvršenje i vraćati vrijednosti korak po korak.

```
import kotlinx.coroutines.*

suspend fun fibonacciGenerator(): List<Int> = coroutineScope {
    val result = mutableListOf<Int>()
    var prev = 0
    var curr = 1
    repeat(10) {
        result.add(curr)
        val next = prev + curr
        prev = curr
        curr = next
    }
    result
}

fun main() = runBlocking {
    val fibonacci = fibonacciGenerator()
    println("Prvih 10 Fibonacci brojeva:")
    fibonacci.forEach { println(it) }
}
```

Listing 3.17: Generator uz pomoć korutina

U primjeru 3.17, funkcija `fibonacciGenerator` je suspend funkcija koja koristi korutine. Koristi se `coroutineScope` kako bi se omogućilo korištenje korutina unutar funkcije. U tijelu `coroutineScope` bloka, koristimo `repeat(10)` petlju da bismo generirali prvih 10 Fibonaccijevih brojeva. Svaki Fibonacci broj se dodaje u listu `result`. Nakon petlje, lista se vraća kao rezultat funkcije.

Bibliografija

[1] https://en.cppreference.com/w/cpp/language/coroutines.

[2] https://lewissbaker.github.io/2017/11/17/
understanding-operator-co-await.

[3] https://github.com/andreasbuhr/cppcoro.

[4] Josuttis M. N., *C++20 - The Complete Guide*, Leanpub.com, 2022.

Sažetak

U ovom diplomskom radu bavimo se korutinama i asinkronim programiranjem. Prvenstveno opisujemo rad korutina u programskom jeziku C++ i implementiramo primjere za bolje upoznavanje asinkrone paradigme.

Rad je započet detaljnim opisivanjem standardne implementacije korutina u programskom jeziku C++ i biblioteke <coroutine> uz pomoć koje su korutine standardizirane 2020. godine. Kako biblioteka ne pruža potpunu podršku za rad s korutinama pokazujemo od nule implementaciju dodatnih klasa koje su potrebne, a to su: sučelje za korutine, promise tip i *Awaitable* tip. Kroz poglavlje unaprijeđujemo implementaciju klasa i pokazujemo kako klase međusobno surađuju za što bolju manipulaciju korutinama. Osim standardnih asinkronih primjera, prikazujemo implementaciju generatora uz pomoć korutina.

U drugom poglavlju pokazujemo open source biblioteku `cppcoro` s kojom korutine možemo koristiti na apstraktniji način te ih ne koristimo kao elemente niske razine što je viđeno u prvom poglavlju. Ne ulazimo u implementaciju same biblioteke nego opisujemo elemente koja one posjeduje i njihovo korištenje. Implementiramo primjere u kojima se vidi korisnost alata u suočavanju s asinkronim programiranjem. Također prikazujemo generatore koje biblioteka sadržava implementirane kao sučelja za korutinu. Na kraju tog poglavlja prikazujemo kako se biblioteka suočava s umrežavanjem (eng. *networking*) te komentiramo koje protokole podržava i na koji način.

Tema zadnjeg poglavlja su korutine u drugim programskim jezicima. Prikazujemo njihovo korištenje u Pythonu, JavaScriptu i Kotlinu te opisujemo kakvu podršku imaju u usporedbi sa C++. Za spomenute programske jezike postoji dobra podrška za korutine te nije potrebna dodatna implementacija za korištenje korutina. U Pythonu, uz ključne riječi `async` i `await`, koristi se biblioteka `asyncio` koja olakšava asinkrono programiranje. U JavaScriptu se pojavljuju iste ključne riječi, a upravljanje korutinama riješavamo uz pomoć `Promise` objekata koji se manipuliraju uz pomoć `Bluebird` biblioteke. Korutine u Kotlinu se definiraju na malo drugačiji način, a za njihovo bolje upravljanje koristimo biblioteku `kotlinx.coroutines`.

Summary

In this master's thesis, we focus on coroutines and asynchronous programming. Primarily, we describe the functioning of coroutines in the C++ programming language and implement examples to better understand the asynchronous paradigm.

The work begins with a detailed description of the standard coroutine implementation in the C++ programming language and the <coroutine> library, which standardized coroutines in 2020. Since the library does not provide complete support for working with coroutines, we demonstrate the implementation of additional necessary classes from scratch, namely: coroutine interface, promise type, and Awaitable type. Throughout the chapter, we improve the implementation of these classes and illustrate how they collaborate for better coroutine manipulation. In addition to standard asynchronous examples, we demonstrate the implementation of generators using coroutines.

In the second chapter, we showcase the open-source library `cppcoro`, which allows us to use coroutines in a more abstract manner, rather than as low-level elements as seen in the first chapter. We do not delve into the implementation of the library itself but describe its elements and their usage. We implement examples that highlight the utility of the tool in dealing with asynchronous programming. Additionally, we showcase generators implemented as coroutine interfaces within the library. Towards the end of this chapter, we illustrate how the library handles networking and comment on the protocols it supports and how.

The theme of the final chapter is coroutines in other programming languages. We demonstrate their usage in Python, JavaScript, and Kotlin, describing the support they have compared to C++. For the mentioned programming languages, there is robust support for coroutines, eliminating the need for additional implementation to use them. In Python, alongside the keywords `async` and `await`, the `asyncio` library facilitates asynchronous programming. In JavaScript, similar keywords are used, and coroutine management is handled using `Promise` objects manipulated with the `Bluebird` library. Coroutines in Kotlin are defined in a slightly different manner, and we use the `kotlinx.coroutines` library for their better management.

Životopis

Rođen sam 11. prosinca 1999. godine u Čakovcu. Osnovnu školu sam završio u Gornjem Mihaljevcu 2014. godine te sam nakon nje upisao prirodoslovno-matematički smjer u Gimnaziji Josipa Slavenskog Čakovec. Spomenutu srednju školu sam završio 2018. godine. Iste godine sam nastavio svoje školovanje u Zagrebu na zagrebačkom Prirodoslovno – matematičkom fakultetu. Upisao sam preddiplomski studij Matematika na matematičkom odsjeku koji sam završio 2021. godine. Na jesen iste godine sam upisao diplomski studij Računarstvo i matematika.