

Korutine u C-u

© 2005 Željko Vrba

“Klasičan” način strukturiranja programa su potprogrami¹ koji se baziraju na sljedećem principu: funkcija ima *točno jedan* ulaz te jedan ili više izlaza.² Prije ulaza u funkciju se specificiraju argumenti, a prilikom izlaza funkcija može pozivajućem kodu vratiti neku vrijednost (najčešće samo jednu, mada moderniji jezici dopuštaju vraćanje više vrijednosti).

Ono što C ne podržava³ jest *prekid izvršavanja* neke funkcije i *njeno naknadno nastavljanje od točke zaustavljanja*. Neki jezici to podržavaju u slabijoj formi (npr. generatori u Pythonu i ključna riječ `yield`), a neki u potpunosti (npr. Scheme i `call-with-current-continuation`).

1	Mehanizam poziva u C-u	1
2	Kontrola toka poziva u C-u	2
2.1	Poziv funkcije i return	2
2.2	setjmp i longjmp	3
2.3	context funkcije	5
3	Implementacija libraryja za korutine	9
3.1	Implementacijski detalji	9
3.2	Interface	9
4	Reference	11

1 Mehanizam poziva u C-u

Ovdje će se opisati mehanizam poziva funkcija u C-u te uvesti neki termini koji su bitni za razumijevanje ostatka teksta. Iako C standard ne zhatijeva upotrebu stacka za prenošenje argumenata između funkcija, to je najčešća implementacija iz dva razloga:

1. Stack je direktno hardverski podržan od većine procesora raspoloživih na tržištu.
2. Stackom se elegantno rješava problem rekurzivnih funkcija.

Promotrimo deklaraciju i poziv neke funkcije:⁴

```
unsigned int eqlen(const char *a, const char *b)
{
    unsigned int r = 0;

    while((*a && *b) && (*a++ == *b++)) r++;
    return r;
}
```

¹ U nastavku će se koristiti isključivo naziv “funkcija”. Neki jezici, npr. Pascal i njegovi derivati razlikuju procedure i funkcije, dok drugi, npr. C i njegovi derivati imaju samo pojam funkcije.

² Ovdje se pod “izlaz” misli točka u kodu gdje se funkcija vraća odakle je pozvana, a *ne* na povratnu vrijednost funkcije. Iako pravila “lijepog” strukturiranog programiranja nalažu da funkcija treba imati i samo *jedan izlaz*, u praksi se često nailazi na slučajeve kada bi takav stil programiranja činio kod *nepreglednijim* i težim za održavanje.

³ na razini jezika

⁴ Radi objašnjavanja principa se namjerno zadržavam na jednostavnom primjeru sa primitivnim tipovima. Stvar se malo komplicira kad funkcija vraća neku strukturu. Obično tada kompajler dodaje jedan argument (nevidljivo programeru) koji sadrži adresu strukture rezultata. U pozivajućoj funkciji također alocira `auto` varijablu istog tipa koja će primiti rezultat funkcije.

```

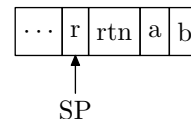
void f()
{
    unsigned int r = eqlen("abbde", "abcdefg");
}

```

Na [slici 1](#) je prikazan sadržaj procesorskog stack-a neposredno nakon poziva funkcije i prije izvršavanja prve naredbe (to se obično naziva funkcijski *stack frame*).

On nastaje tako što kompajler generira sljedeći kod:

1. Stavlja na stack vrijednosti pointera `b` i `a`.⁵
2. Poziva funkciju `eqlen` instrukcijom poput npr. `CALL`. `rtm` je povratna adresa koju procesor pri pozivu potprograma `CALL` instrukcijom automatski stavlja na stack. Ta adresa pokazuje na instrukciju *neposredno nakon* `CALL` instrukcije.
3. Alocira prostor za lokalne varijable funkcije (obično jednostavnim *oduzimanjem*⁶ ukupne veličine lokalnih varijabli od trenutne vrijednosti stack pointera).
4. Inicijalizira vrijednosti lokalnih varijabli kojima je zadana početna vrijednost.



Slika 1 Stack frame `eqlen` funkcije

Iz ovoga se vidi i zašto nije definirana vrijednost neinicijaliziranih lokalnih varijabli: one jednostavno poprime vrijednost koja se u tom trenutku našla na stacku.

Napomena: ovo je jedan mogući izgled stack framea. U praksi kompajler, obično između povratne adrese (`rtm`) i prve lokalne varijable (`r`), po potrebi sprema vrijednosti nekih registara. No to je već ulaženje u detalje tzv. calling konvencija i to je tema za neki drugi članak.

Pri povratku iz funkcije dešava se sljedeće:

1. Dealocira se prostor koji su zauzimale lokalne varijable (sada je to *dodavanje* veličine lokalnih varijabli trenutnoj vrijednosti stack pointera).
2. Izvršava se `RET` instrukcija. Procesor uzima adresu sa vrha stacka i nastavlja izvršavanje koda od te adrese.
3. Pri povratku u pozivajuću funkciju, čisti se prostor na stacku koji su zauzimali argumenti funkcije.
4. Povratna vrijednost funkcije se najčešće vraća u nekom registru.

2 Kontrola toka poziva u C-u

2.1 Poziv funkcije i return

Ovaj mehanizam je već opisan u prethodnom poglavlju. `return` naredba može se pojaviti bilo gdje unutar tijela funkcije.

⁵ Argumenti se (obično) stavljaju *obrnutim* redoslijedom jer to jako olakšava rad sa funkcijama koje imaju promjenjiv broj argumenata.

⁶ Stack obično “raste” prema dolje, tj. stavljanjem elemenata na stack, stack pointer pokazuje na sve *manje* memorijske lokacije.

2.2 setjmp i longjmp

Sam prekid izvršavanja je podržan C library funkcijama `setjmp` i `longjmp` koje su deklarirane u headeru `<setjmp.h>`. Te funkcije podržavaju tzv. “non-local goto”. Naravno, `return` naredba uvijek prekida izvršavanje trenutne funkcije, ali se uvijek vraća pozivajućoj funkciji – nema mogućnosti “preskakanja” nekoliko nivoa poziva.

`setjmp` i `longjmp` podržavaju povratak *nekoliko* razina poziva “iznad” i eksplicitno zabranjuju skok u kontekst funkcije koja je završila. Ako se to ipak napravi, rezultat je *nedefiniran* i najčešće rezultira “rušenjem” programa.

`setjmp` funkcija čuva trenutnu lokaciju izvršavanja programa u varijabli tipa `jmp_buf`. Pri prvom povratku vraća vrijednost 0, a pri svakom sljedećem povratku vraća vrijednost koja je drugi argument funkcije `longjmp`.

`setjmp` ne čuva vrijednosti svih registara te pri povratku (nakon poziva `longjmp` funkcije) vrijednosti lokalnih varijabli koje su bile spremljene u registrima ne moraju biti vraćene na ispravne vrijednosti. Zbog toga treba sve⁷ lokalne varijable koje će se koristiti nakon povratka iz `setjmp` deklarirati kao *volatile*.

Sljedeći primjer ilustrira mehanizam rada `setjmp` i `longjmp` i usput pokazuje kako se elegantno može prekinuti vraćanje iz duboke rekurzije.⁸

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

static jmp_buf jbuf;

struct node {
    int x;
    struct node *next;
};

static struct node *populate_list(int n)
{
    struct node *list = NULL;

    while(n > 0) {
        struct node *e = malloc(sizeof(struct node));
        e->next = list;
        e->x = n--;
        list = e;
    }

    return list;
}
```

⁷ Zapravo, ne baš sve – postoje iznimke. Jedna iznimka je varijabla koja se koristi za čuvanje povratne vrijednosti `setjmp` funkcije. Pravila su relativno složena i za točan opis je najbolje pogledati C standard.

⁸ Ovo se može puno jednostavnije iskodirati, ali je ilustrativno kao primjer.

```

static int
find1(struct node *l, int a, int current)
{
    int r;
    if(!l)
        return -1;
    if(l->x == a)
        return current;
    r = find1(l->next, a, 1+current);
    printf("vracam %d\n", r);
    return r;
}

int main(void)
{
    struct node *l;
    int n, a, p;

    printf("n a="); scanf("%d %d", &n, &a);
    l = populate_list(n);

    p = find1(l, a, 0);
    printf("find1=%d\n", p);

    if(setjmp(jbuf) == 0) {
        p = 0;
        find2(l, a, &p);
    } else {
        printf("find2=%d\n", p);
    }

    return 0;
}

static void
find2(struct node *l, int a, int *current)
{
    if(!l) {
        *current = -1;
        longjmp(jbuf, 1);
    }
    if(l->x == a)
        longjmp(jbuf, 1);
    ++*current;
    find2(l->next, a, current);
}

```

Program generira listu od n elemenata ($1 \dots n$), a funkcije `find1` i `find2` traže poziciju elementa a u listi. Obje su napisane rekurzivno na vrlo sličan način. Razlike koje treba uočiti su sljedeće:

- Kada funkcija `find1` vraća rezultat k , onda k puta vraća jedno te istu vrijednost (svim rekurzivnim pozivima) što je bespotrebno utrošeno vrijeme.
- U općem slučaju `longjmp` ne može vratiti proizvoljnu vrijednost. Zbog toga `find2` uzima varijablu `current` kao pointer i koristi ju za povratnu vrijednost. To se reflektira i na pozivu `find2` funkcije gdje se `p` prvo inicijalizira na 0 (što je preduvjet za ispravan rad funkcije).
- `find2` se “vraća” samo jedanput sa izračunatom vrijednošću.

U ovom primjeru je korištena globalna varijabla `jbuf` za “označavanje” mjesta kamo se `longjmp` treba vratiti. Kako to općenito nije dobra praksa, za vježbu napišite gornji program *bez* korištenja globalne varijable, ali uz promijenjen prototip `find2` funkcije:

```
static void find2(jmp_buf jbuf, struct node *l, int a, int *current);
```

Tada varijabla `jbuf` može biti lokalna u `main` funkciji.⁹

Iako je ovo prejednostavan problem da bi se u praksi ovako rješavao (jer se može riješiti isključivo iterativno – napravite to za vježbu!), prvi idući koji bi se isplatilo ovako optimizirati je pretraživanje binarnog stabla.

Pandan `setjmp` i `longjmp` funkcijama su exceptioni u drugim programskim jezicima. Pri korištenju ovih funkcija sa C++ jezikom treba paziti jer *destruktori objekata na stacku ne moraju biti pozvani* (funkcije su i dalje “sigurne” za korištenje ako svi objekti na stacku u lancu poziva funkcija nemaju user-defined destruktor).

2.3 context funkcije

Ove funkcije postoje na svim modernim UNIX operacijskim sustavima¹⁰. To je zapravo familija sljedećih funkcija: `getcontext`, `setcontext`, `makecontext` i `swapcontext`. Da bi ih se koristilo treba uključiti `<ucontext.h>` header. Točan način rada tih funkcija se neće opisivati jer su one detaljno opisane u man stranicama. Umjesto toga će se na primjeru pokazati kako se i u C-u mogu simulirati kontinuirane kao u Scheme programskom jeziku. Za primjer je uzet inorder obilazak binarnog stabla traženja.

Sljedeći dio programa je prilično standardan. Implementira alokaciju čvora i inicijalizaciju nekom vrijednošću (funkcija `newnode`), umetanje čvora u stablo (funkcija `bt_insert`) te uobičajeni (rekurzivan) inorder obilazak i ispis čvorova stabla (funkcija `recursive_inorder`) te glavni program koji korisniku dopušta unos nekoliko elemenata u stablo i ispisuje ih rekurzivno i iterativno.

```
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>

struct node {
    int data;
    struct node *left, *right;
};

static struct node *newnode(int data)
{
    struct node *n = malloc(sizeof(struct node));
    n->left = n->right = NULL;
    n->data = data;
    return n;
}

static void bt_insert(struct node **root, int data)
{
    struct node *r = *root;
    struct node **prevlink = root;
```

⁹ Ovakav način kodiranja se ponekad zove continuation-passing style jer funkcija kao argument dobiva mjesto u programu gdje treba nastaviti izvršavanje (kontinuiranje). Naziv continuation dolazi iz programskog jezika Scheme i puno je općenitiji: to je kompletno “zamrznuto” stanje izvršavanja programa u nekom trenutku koje se može po volji “restartati”.

¹⁰ Pojavili su se u SUSv2 standardu. (SUS - Single UNIX Specification)

```

while(r) {
    if(data < r->data) {
        prevlink = &r->left;
        r = r->left;
    } else {
        prevlink = &r->right;
        r = r->right;
    }
}
*prevlink = newnode(data);
}

static void recursive_inorder(struct node *root)
{
    if(root) {
        recursive_inorder(root->left);
        printf("%d ", root->data);
        recursive_inorder(root->right);
    }
}

int main(void)
{
    struct node *tree = NULL;
    int i, n, data;

    printf("number of nodes: "); scanf("%d", &n);
    for(i = 0; i < n; i++) {
        printf("? "); scanf("%d", &data);
        bt_insert(&tree, data);
    }
    recursive_inorder(tree);
    printf("\n");
    iterative_inorder(tree);
    return 0;
}

```

Sljedeća funkcija za inorder obilazak stabla je rekurzivna, ali kontrola toka je tako izvedena da se pozivajućoj funkciji čvorovi vraćaju jedan po jedan (dakle, iterativno iz perspektive pozivajuće funkcije):

```

static void *inorder_cont(
    ucontext_t *me, ucontext_t *caller,
    struct node *root, struct node **current)
{
    if(root) {
        inorder_cont(me, caller, root->left, current);

        *current = root;
        swapcontext(me, caller);
    }
}

```

```

        inorder_cont(me, caller, root->right, current);
    }
}

```

U usporedbi za `recursive_inorder` ima sljedeće razlike:

- Argumenti `me` i `caller` služe za kontrolu toka programa. U trenutku poziva `swapcontext` funkcije se trenutno stanje izvođenja funkcije sprema u varijablu `me`, a izvršavanje se nastavlja na mjestu koje je spremljeno u varijabli `caller`. Kada pozivajuća funkcija ponovno vrati kontrolu ovoj funkciji, izvršavanje se nastavlja neposredno nakon poziva `swapcontext` funkcije (izgleda kao da se `swapcontext` funkcija “vratila”).
- Postoji isti problem sa vraćanjem vrijednosti kao i sa `longjmp`. Zbog toga je ovdje dodatan argument `current` preko kojega se vraća pointer na trenutni čvor koji funkcija posjećuje.

Dodjeljivanje varijabli `current` i poziv `swapcontext` funkcije zajedno čini “vraćanje vrijednosti” trenutnog čvora pozivajućoj funkciji.

Funkcija `iterative_inorder` ima isti prototip kao i `recursive_inorder` ali su način rada i implementacija bitno drugačiji.

```

1  static void iterative_inorder(struct node *root)
2  {
3      volatile int firsttime = 1;
4      ucontext_t here, pre, iterate;
5      char stack[8192];
6      struct node *current;

7      getcontext(&here);          /* # */
8      if(!firsttime) {
9          printf("End.\n");
10         return;
11     }
12     firsttime = 0;

13     /* inimizijalizacija contexta za inorder() funkciju */
14     getcontext(&pre);
15     pre.uc_link = &here;        /* vracamo se u # kad f-ja zavrsi */
16     pre.uc_stack.ss_sp = stack;
17     pre.uc_stack.ss_size = sizeof(stack);
18     makecontext(&pre, inorder_cont, 4, &pre, &iterate, root, &current);

19     /* iterativni obilazak stabla */
20     while(1) {
21         swapcontext(&iterate, &pre);
22         printf("%d ", current->data);

23         /* primjer prekidanja rekurzije ako je data 0 */
24         if(!current->data) {
25             printf("Ending traversal: 0 encountered.\n");
26             break;
27         }

```

```
28     }
29 }
```

Linije 3-6 deklariraju sve korištene lokalne varijable:

- `firsttime` je varijabla koja se pri prvom ulasku u funkciju inicijalizira na 1. Deklarirana je kao `volatile` da kompajler izbjegne bilo kakvu optimizaciju sa spremanjem u registar.
- `here` kontekst se koristi za vraćanje u ovu (`iterative_inorder`) funkciju kada se obiđe cijelo stablo.
- `pre` i `iterate` konteksti se koriste za “prebacivanje” kontrole toka između ove (`iterate` kontekst) i `inorder_cont` funkcije (`pre` kontekst).
- `stack` je stack na kojem će se funkcija `inorder_cont` izvršavati. Kao što je u početku napisano, stack je vrlo bitan za izvršavanje funkcije i sadrži povratnu adresu, argumente i lokalne varijable (dakle, kompletno stanje funkcije). Mehanizam korutina je omogućen upravo time što funkcije *ne koriste zajednički stack*. Općenito su bitne dvije stvari:
 - Stack mora biti dovoljno velik. Koliko je “dovoljno” ovisi o broju argumenata funkcije, veličini lokalnih varijabli, potrebama za stackom ostalih pozvanih funkcija, itd. Ako nije dovoljno velik, program će se srušiti.
 - Ovdje je stack stavljen kao lokalna varijabla. Ono što je bitno jest da adresa stacka mora biti valjana sve dok postoji mogućnost “povratka” u neki kontekst sa `swapcontext` ili `setcontext`. Zbog toga je općenito preporučljivo alocirati stack na heapu i *osloboditi ga tek kad se neki kontekst uništi*.

Linije 7-11 služe za izlazak iz funkcije ako `inorder_cont` završi: naime, (kako će biti objašnjeno kasnije) tada se kontrola toka vraća upravo iza mjesta poziva `getcontext` funkcije. Da bi se spriječilo beskonačno izvršavanje jedno te iste funkcije, ispituje se vrijednost varijable `firsttime` i ako je 0, izlazi se van iz funkcije.

U 12. liniji se postavlja varijabla `firsttime` kako bi upravo opisani mehanizam završetka funkcije ispravno radio.

Linije 13-18 uspostavljaju kontekst za poziv `inorder_cont` funkcije. Prvo se `pre` kontekst mora inicijalizirati pozivom `getcontext` funkcije. Nakon toga se postavlja polja `ucontext_t` strukture:

- `uc_link` je pointer na kontekst gdje će se nastaviti izvršavanje ako `pre` kontekst završi. Postavlja se tako da se vraća upravo na liniju označenu sa `#`. Ako je ovo polje `NULL`, onda se prekida izvršavanje programa.
- Linije 16 i 17 postavljaju stack (adresu i veličinu) na kojem će se kontekst izvršavati

`makecontext` funkcija inicijalizira kontekst tako da iduća aktivacija konteksta (`setcontext` ili `swapcontext` funkcijom) počinje izvršavanje od zadane funkcije. Prvi argument je kontekst koji se inicijalizira, drugi argument je pointer na funkciju, treći argument je broj argumenata funkcije (u našem slučaju 4), a ostali argumenti će se prenijeti `inorder_cont` funkciji.

Linije 19-29 iterativno ispisuju čvorove stabla. `swapcontext` funkcija će pri prvom pozivu “pozvati” funkciju `inorder_cont`, a pri svakom sljedećem pozivu će se “vratiti” na mjesto poziva `swapcontext` u `inorder_cont` funkciji.

Obratno, poziv `swapcontext` u `inorder_cont` funkciji se “vraća” na mjesto poziva `swapcontext` u `iterative_inorder`. Korutina vraća vrijednost preko `current` pointera.

Ključno je shvatiti na koje konkretne kontekste se odnose argumenti poziva `swapcontext` funkcije na oba mjesta!

Dakle, funkcije kao da se “međusobno” pozivaju i na neki način su ravnopravne – otuda i naziv korutina (engl. *coroutine*; klasični potprogram je engl. *subroutine*).

Kako bi se prikazale prednosti do sada opisanih tehnika kontrole toka, kao primjer je u kod uključen uvjet koji prekida ispisivanje stabla čim se nađe element koji je jednak nuli. Za vježbu pokušajte modificirati funkciju `recursive_inorder` (bez korištenja `setjmp/longjmp` funkcija!) da prekine ispisivanje stabla čim nađe element jednak nuli.

3 Implementacija libraryja za korutine

Kao što je se može vidjeti iz prethodnog primjera, “sirove” context funkcije su prilično nespretne za korištenje. Ovdje će se opisati ideja implementacije jednostavnog libraryja koji bitno olakšava korištenje korutina. U referencama se nalazi URL do source koda svih primjera iz članka te libraryja.

3.1 Implementacijski detalji

Prvo se definira struktura korutine i deklariraju pomoćne funkcije. Treba napomenuti da ovu strukturu i funkcije “klijentska aplikacija” mora smatrati *implementacijskim detaljom* i ne smije ih direktno pozivati, mijenjati članove strukture niti se oslanjati na njihov sadržaj. Aplikacija treba koristiti isključivo makroe iz *javnog interfeasa*.

```
struct co_coroutine {
    ucontext_t me, caller, join;
    void *stack, *result;
    size_t stack_size;
    int finished;
};

void co_init(struct co_coroutine *co, size_t stack_size);
void co_free(struct co_coroutine *co);
```

`co_coroutine` struktura sadrži potreban skup podataka za laku kontrolu toka:

- `me` je kontekst korutine, `caller` je kontekst funkcije koja je pozvala korutinu, a `join` je kontekst funkcije koja će se izvršiti kada korutina završi.
- `stack` je pointer na stack korutine, a `stack_size` je alocirana veličina stacka.
- `result` uvijek pokazuje na lokaciju kamo korutina sprema rezultat.
- `finished` se inicijalizira na 0. Kada korutina završi, kontrola se prebacuje na funkciju u `join` kontekstu koja postavi ovu varijablu na 1 te prebaci kontrolu toka na pozivajuću funkciju.

`co_init` funkcija alocira stack za korutinu i inicijalizira polja strukture. Treba napomenuti da alocira `JOIN_STACK_SIZE` (konstanta definirana u `coroutine.c`) *više* mjesta nego je specificirano u `stack_size` argumentu i taj dodatni prostor služi za izvršavanje `co_join` funkcije.

`co_free` funkcija oslobađa stack korutine i postavlja na 0 strukturu. To olakšava otkrivanje grešaka zato što će se program srušiti čim program pokuša ući u korutinu koju je oslobodio.

3.2 Interface

Interface libraryja je detaljno dokumentiran u `coroutine.h` i čine ga jedan tip (`co_coroutine`) i četiri makroa.

CO_CREATE makro ima sljedeću definiciju:

```
#define CO_CREATE(co, stack_size, func, nargs, ...) \
    (co_init(co, stack_size), \
     makecontext(&*(co).me, (void(*)())func, 1+nargs, co, __VA_ARGS__), \
     *(co).stack != NULL)
```

Treba obratiti pažnju na tri detalja:

1. Elementi makroa su odvojeni zarezom, što znači da kompletan makro čini jedan *izraz* koji vraća vrijednost posljednjeg elementa u listi. Ako je povratna vrijednost 0, korutina nije kreirana (jer nije uspjela alokacija memorije za stack).
2. Pristup elementima strukture preko co pointera ne koristi -> sintaksu, već nečitkiju formu, npr: `*(co).me` umjesto `co->me`. To je kako bi se spriječile sintaksne (i semantičke) greške nakon ekspanzije makroa.
Npr. kada bi se pisalo `co->me` i makro se pozove sa `CO_CREATE(&co, 8192)`, tada bi se prvi argument `makecontext` funkcije “expandao” na `& & co->me` (prvi `&` je u ekspanziji makroa, drugi dolazi kao dio argumenta makroa) što je sintaksna greška.
3. `makecontext` izgrađuje kontekst korutine sa *jednim argumentom više*: prvi argument je upravo pointer na korutinu i u trenutku izvršavanja korutina preko prvog argumenta može pristupiti podacima o sebi. Dakle, funkcija koja će se izvršavati kao korutina **mora** imati sljedeći prototip:

```
void coroutine_function(co_coroutine *co, ...);
```

(gdje ... *ne* označava nužno variadic funkciju – programer može staviti argumente kako mu odgovara).

Par CO_CREATE makrou je CO_FREE makro koji ima trivijalnu definiciju i stavljen je radi konzistentnosti interfeaca.

CO_RESUME i CO_YIELD makroi stvarno prenose kontrolu toka programa i djeluju u paru te ih tako treba i promatrati:

```
#define CO_RESUME(co, rvar) \
    ((*co).result = rvar, swapcontext(&*(co).caller, &*(co).me), \
    !*(co).finished)
```

```
#define CO_YIELD(co, rvar) do { \
    memcpy((*co).result, rvar, sizeof(*rvar)); \
    swapcontext(&*(co).me, &*(co).caller); \
} while(0)
```

CO_RESUME makro se ponaša kao funkcija, upravo kao i CO_CREATE:

1. Inicijalizira `result` pointer korutine – na to mjesto će korutina spremi rezultat.
2. `swapcontext` funkcijom prebacuje kontrolu na korutinu. Trenutni kontekst izvršavanja se sačuva u `caller` polju, a kontrola se prebacuje na kontekst definiran `me` poljem (i upravo taj kontekst je inicijaliziran u CO_CREATE makrou).
3. Kada se korutina vrati (pozivom CO_YIELD makroa), izračunava se vrijednost `finished` polja što je ujedno i rezultat koji makro vraća.

Treba napomenuti da je ponovni poziv korutine nakon što ovaj makro vrati 0¹¹ *programerska greška* i program će završiti porukom o grešci i pozivom `abort()` funkcije.

Primijetite da se ovaj makro ponaša kao funkcija, upravo kao i `CO_CREATE`.

`CO_YIELD` makro prekida izvršavanje korutine i vraća vrijednost pozivajućoj funkciji:

1. `co` je pointer na trenutnu korutinu – to je upravo prvi `i`, kao što je ranije već rečeno, obavezan prvi argument funkcije korutine.
2. `rvar` je pointer na varijablu koja sadrži povratnu vrijednost i `memcpy` kopira njenu vrijednost u `result` pointer korutine. No taj pointer je inicijaliziran u `CO_RESUME` makrou neposredno prije nego što je kontrola predana korutini.
3. `swapcontext` funkcijom se vraća pozivajućoj funkciji.

4 Reference

1. Portabilan paket korutina za C++ koji se ne oslanja na sistemske pozive. Ima i puno drugih primjera gdje je zgodno koristiti korutine.
<http://www.akira.ruc.dk/~keld/research>
2. Python uvodi “slabiju” verziju korutina u vidu generatora i ključne riječi `yield`.
<http://www.python.org>
3. Scheme (derivat LISP-a) ima podršku za potpune korutine (zване kontinuuacije). Na navedenom linku se može naći dokumentacija i implementacije.
<http://www.schemers.org>
4. Ruby i Lua su elegantni jezici koji također podržavaju potpune korutine.
<http://www.ruby-lang.org/en>
<http://www.lua.org>
5. GNU PTH je user-level implementacija threadova za UNIX operacijske sustave. Osim simulacije threadova, također implementira ekvivalente `*context` funkcija koje se mogu koristiti na sustavima koji nemaju native `*context` funkcije.
<http://www.gnu.org/software/pth>
6. Još jedna implementacija korutina u “čistom” C-u.
<http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>
7. Enciklopedijsko objašnjenje korutina i kontinuuacija uz daljnje linkove.
<http://c2.com/cgi/wiki?ContinuationsAndCoroutines>

¹¹ u tom slučaju se kontrola nastavlja nakon `swapcontext` u `co_join`

