Vežba 3 - Pravila pisanja koda u programskom jeziku C

1. Cilj

U okviru ovog PDF-a predstavljen je preporučeni stil pisanja koda u programskom jeziku C, koji će biti korišćen tokom trajanja ovog kursa. Kod pisan "dobrim stilom" jeste onaj kod koji je:

- Organizovan
- Lako čitljiv
- Lako razumljiv
- Lak za održavanje
- Efikasan

2. Organizacija datoteke

Osnovnu strukturu svake datoteke čine:

- Prolog datoteke, koji može uključiti i opis implementiranog algoritma
- Upotreba i način upotrebe datoteke
- Zaglavlja
- Makroi i definicije tipova koji se koriste u celoj datoteci
- Spoljašnje promenljive
- Sekvenca funkcija koje se sastoje od prologa i tela

2.1. Prolog datoteke

Prolog datoteke predstavlja skup relevantih podataka o datoteci. Primeri prologa za zaglavlja i izvorne datoteke su dati u nastavku.

```
/**
 * @file ime_datoteke.c
 * @brief Opis datoteke
 * @author Ime Prezime
 * @date DD-MM-YYYY
 * @version 1.0
 */
```

```
/*****************************************************************
 * NAME:                                                         *
 *                                                               *
 * PURPOSE:                                                      *
 *                                                               *
 * GLOBAL VARIABLES:                                             *
 *                                                               *
 * Variable          Type      Description                       *
 * --------          ----      -----------                       *
 *                                                               *
 * DEVELOPMENT HISTORY:                                          *
 *                                                               *
 * Date     Author   Change Id   Release    Description Of Change *
 * ----     ------   ---------   -------    ---------------------- *
 *                                                               *
 *****************************************************************/
```

```
/*****************************************************************
 * FILE NAME:                                                    *
 *                                                               *
 * PURPOSE:                                                      *
 *                                                               *
 * FILE REFERENCES:                                              *
 *                                                               *
 * Name                        I/O         Description           *
 * ----                        ---         -----------           *
 *                                                               *
 * EXTERNAL VARIABLES:                                           *
 * Source:  <          >                                         *
 *                                                               *
 * Name       Type             I/O         Description           *
 * ----       ----             ---         -----------           *
 *                                                               *
 * EXTERNAL REFERENCES:                                          *
 *                                                               *
 * Name                        Description                       *
 * ----                        -----------                       *
 *                                                               *
 * ABNORMAL TERMINATION CONDITIONS, ERROR AND WARNING MESSAGES:  *
 *                                                               *
 * ASSUMPTIONS, CONSTRAINTS, RESTRICTIONS:                       *
 *                                                               *
 * NOTES:                                                        *
 *                                                               *
 * REQUIREMENTS/FUNCTIONAL SPECIFICATIONS REFERENCES:            *
 *                                                               *
 * DEVELOPMENT HISTORY:                                          *
 *                                                               *
 * Date    Author    Change Id    Release    Description Of Change *
 * ----    ------    ---------    -------    ---------------------- *
 *                                                               *
 * ALGORITHM (PDL)                                               *
 *                                                               *
 *****************************************************************/
```

2.2. Uključivanje zaglavlja

Upotrebom preprocesorske direktive *#include*, datoteke zaglavlja se uključuju u izvorne datoteke. Kako bi obezbedili najveću efikasnost i preglednost, potrebno je uključiti samo potrebne datoteke zaglavlja. Ukoliko razlog uključivanja nije očigledan, potrebno je komentarom naznačiti zbog čega je to učinjeno.
Uključivanje datoteka zaglavlja izvršava se sledećim redosledom:

1. Standardne biblioteke (*stdio.h* i *stdlib.h*)
2. Ostale sistemske biblioteke
3. Korisničke datoteke zaglavlja

2.3. Makroi i definicije tipova

Nakon uključivanja datoteka zaglavlja, definišu se konstante, tipovi i makroi koji će se upotrebljavati u nastavku datoteke. Definisanje se vrši sledećim redom:

1. Enumeracija - definisanje enumerisanog tipa podataka
2. Definisanje tipova podataka
3. Definisanje konstanti upotrebom makroa (*#define identifier token-string*)
4. Definisanje funkcionalnih makroa (*#define identifier(identifier, ..., identifier) token-string)*)

2.4. Sekvenca funkcija

U ovom poglavlju data su pravila raspoređivanja funkcija unutar izvorne datoteke, dok će pravila organizacije unutar funkcija biti prikazana u narednim poglavljima. Pre definisanja funkcije, obično se navodi prolog koji daje objašnjenje funkcije sa stanovišta funkcionalnosti, ulaznih parametara i povratne vrednosti. Neka od pravila koje treba primeniti prilikom organizovanja funkcija su:

- Ukoliko datoteka sadrži funkciju *main()*, tada ova funkcija treba da bude prva u datoteci.
- Potrebno je smestiti logički povezane funkcije unutar iste datoteke
- Za raspoređivanje funkcija preferira se *breadth-first* pristup, tj. navođenje funkcija po nivoima apstrakcije tako da su grupisane funkcije istog nivoa.
- Ukoliko se definiše veliki broj nezavisnih osnovnih funkcija, potrebno ih je navesti po abecednom redu.
- Kako bi se povećala preglednost, potrebno je razdvojiti funkcije jednim redom zvezdica.

```
/*****************************************************/

main prolog
main body

/*****************************************************/

function_a prolog
function_a body

/*****************************************************/

function_b prolog
function_b body

/*****************************************************/
```

3. Organizacija funkcija

Standardna funkcija se sastoji od sledećih delova:

- Prolog funkcije
- Deklaracije argumenata funkcije
- Deklaracije promenljivih
- Sekvence paragrafa naredi
- *Return* naredbe

3.1. Prolog funkcije i deklaracija argumenata

Svaka funkcija treba da poseduje prolog čija je uloga da upozna čitaoce sa funkcijom. Prolog funkcije treba da sadrži:

- Ime funkcije i kratak opis
- Listu argumenata, gde je svaki naveden u posebnoj liniji, sa tipom, oznakom da li je ulazni ili izlazni i kratkim opisom
- Tip i opis povratne vrednosti

```
/**
 * PLS7readDisplay - funkcija vraca simbol na zadatom 7SEG
 * displeju ili raspored LED dioda.
 *
 * @param display - In - Selektuje 7SEG ili diode
 * @return 8-bitni simbol na zadatom displeju ili raspored LED dioda
 */
unsigned char PLS7readDisplay(unsigned char display);
```

3.2. Definisanje promenljivih

Prilikom definisanja promenljivih prvo se definišu spoljašnje pa tek nakon njih unutrašnje (lokalne) promjenljive. Lista pravila koje treba koristiti prilikom definisanja lokalnih promenljivih su:

- Poravnati definicije lokalnih promenljivih tako da se prvo slovo svake promenljive nalazi u istoj koloni
- Definisati svaku promenljivu u posebnoj liniji koju prati po potrebi opisni komentar. Jedini izuzetak je definicija iteracionih promenljivih koje mogu biti deklarisane u istom redu.
- Ako grupa funkcija koristi isti parametar ili promenljivu, nazvati ponovljenu promenljivu istim imenom
- Izbegavati redefinisanje promenljivih definisanih na višem nivou lokalnim promenljivama.

3.3. Paragrafi naredbi

Nakon definisanja promenljivih, praznim redom se odvajaju delovi funkcije koji se sastoje od logički grupisanih naredbi, kako bi se obezbedila bolja preglednost. Dodatno se mogu upotrebiti komentari u liniji ili blok komentari kako bi se objasnili određeni delovi funkcije. Blok komentari se koriste pre grupe naredbi, gde oni daju opis algoritma ili rešenja problema, a ne onoga što je iz koda očigledno.

3.4. *Return* naredba

Naredba *return* služi za povraćaj vrednosti iz funkcije onome što je poziva. Prilikom pozivanja naredbe *return*, povratna vrednost može biti rezultat nekog izraza. Slede pravila upotrebe *return* naredbe.

- Upotreba izraza u *return* naredbi poboljšava efikasnost koda, međutim može smanjiti preglednost, zbog čega treba voditi računa da izraz ne bude previše komplikovan.
- Ne treba stavljati više *return* naredbi unutar neke funkcije, sem u slučaju kada nije moguće drugačije implementirati algoritam.
- Uvek deklarisati tip povratne vrednosti funkcije
- Postavljanje samo jedne *return* naredbe u funkciji pravi jednu poznatu tačku kroz koju će se proći prilikom kraja izvršavanja funkcije. Zbog ovoga, izmena povratne vrednosti je jednostavnija.

4. Pravila koja se odnose na čitljivost koda

4.1. Enkapsulacija i sakrivanje informacija

Neka od osnovnih pravila pisanja koda su:

- Enkapsulacija - koja se bazira na grupisanju sličnih elemenata. Enkapsulacija se može raditi na svim nivoima hijararhije (npr. grupisanje sličnih datoteka, podela datoteke na deo namenjen promenljivama i deo namenjen funkcijama, organizacija promenljivih i funkcija u logičke grupe)
- Sakrivanje informacija - predstavlja kontrolisanje vidljivosti elemenata programa (npr. svrstavanje povezanih promenljivih i funkcija u isto zaglavlje i uključivanje tog zaglavlja samo gde je potrebno)

4.2. Prazne linije

Pažljivo korišćenje praznih linija između logičkih celina koda značajno povećava čitljivost i razumljivost samog koda. Sledeći kodni segment ilustruje ispravno korišćenje praznih linija. Pri tome, za razdvajanje delova koda, uvek se koristi jedna prazna linija.

```
#define LOWER  0
#define UPPER  300
#define STEP   20

main()  /* Fahrenheit-Celsius table */
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr - 32));
}
```

4.3. Razmaci

Pravilno pisanje razmaka u kodu, takođe značajno utiče na čitljivost koda. Posebnu pažnju potrebno je posvetiti prilikom formatiranja operatora koji se koriste unutar koda. Neka od pravila podrazumevaju:

- Uvek postaviti razmak između rezervisane (ključne) reči i otvorene zagrade. Primer:
  `if (a > b)`
- Prilikom korišćenja operator `->`, `.` i `[]` nije potrebno pisati razmake. Primer:
  `p->m, s.m, a[i]`
- Nije potrebno pisanje razmaka između imena funkcija i liste parametara. Primer:
  `func(a, b)`

- Nakon svakog zareza, potrebno je pisati jedan razmak. Primer:
  ```
  func(a, b, c, d, f)
  ```
- Prilikom pisanja unarnih operatora, nije potrebno pisati razmake. Primer:
  ```
  !p, ++i, *p, &x
  ```
- Prilikom konverzije tipa, potrebno je pisati jedan razmak. Primer:
  ```
  (int) tmp
  ```
- Prilikom pisanja operatora dodele (=), potrebno je uvek pisati jedan razmak. Primer:
  ```
  x = y
  ```
- Nakon pisanja tačka-zareza (;), potrebno je pisati i jedan razmak (ili novi red). Primer:
  ```
  for (int i; i < n; i++)
  ```
- Prilikom pisanja binarnih operatora, potrebno je pisati po jedan razmak sa obe strane. Primer:
  ```
  x + y + z, a < b && b < c
  ```
- Prilikom konverzije tipa, potrebno je pisati jedan razmak. Primer:
  ```
  (int) tmp
  ```
- Koristite zagrade, kako biste naglasili prednost računskih operacija. Ovo je posebno važno prilikom korišćenja *bitwise* operatora (~, &, | i ^) u kombinaciji sa operatorom pomeranja (<<, >>). Primer:
  ```
  PORTB = ~(0x01 << (4-displej));
  ```
- Dugačke uslovne izraze (na primer u *if* naredbama) potrebno je prelomiti u više linija. Primer:
  ```
  if ((total_count < needed) &&
     (needed <= MAX_ALLOC) &&
     (flag == 0))
  {
        statement1;
        statement2;
        statement3;
  }
  ```

4.4. Uvlačenje (Indentacija)

Koristite indentaciju kako biste jasno naglasili logičku strukturu koda. Standardna veličina uvlačenja koja omogućava dobru preglednost je 4 razmaka. Primer dobro uvučenog koda.

```
main()
{
    int c;

    c = getchar();
    while (c!= EOF)
    {
        putchar(c);
        c = getchar();
    }
}
```

4.5. Komentari

Komentari obezbeđuju dodatna objašnjenja delova koda za nekog ko čita kod. Takođe, pomoću komentara se vrši naglašavanje pojedinih sekcija unutar koda. Moguće ih je pisati na više različitih nivoa:

- unutar prologa datoteke
- na nivou funkcija (prolog funkcije)
- kroz sam kod (objašnjenje svrhe promenljivih, izraza itd.)

Razlikuju se sledeće vrste komentara:

- *Boxed* komentari - koriste se prilikom pisanja prologa

```
/*********************************************************
 * FILE NAME                                             *
 *                                                       *
 * PURPOSE                                               *
 *                                                       *
 *********************************************************/
```

- *Block* komentari - koriste se na početku svake veće logičke celine unutar koda (primer: funkcije) za njen opis

```
/*
 *  Write the comment text here, in complete sentences.
 *  Use block comments when there is more than one
 *  sentence.
 */
```

U okviru *Doxygen* alata prolog se piše kao *block* komentar, na sledeći način:

```
/**
 * @file my_library.h
 * @brief My Library
 * @author Name Surname
 * @date 13-03-2021
 */
```

- Kratki (linijski) komentari - koriste se u okviru iste linije koda koja se opisuje ili pre logičke celine koda koja se opisuje u slučaju da je komentar dovoljno kratak da stane u jednu liniju.

```
int max = 7; // Primer komentara koji opisuje promenljivu u istoj liniji
int sum = 7; // Primer komentara koji opisuje promenljivu u istoj liniji

// Primer komentara koji opisuje deo koda u jednoj liniji
for (i = 1; i <= max; i++);
      sum += i;
```

U okviru *Doxygen* alata kratki komentari se pišu na sledeći način.

```
/// Doxygen linijski komentar
#define MAX_VALUE 100
```

4.6. Imena

Generalna pravila prilikom davanja imena fajlovima, funkcijama, konstantama i promenljivima su sledeća:

- koristiti precizna i dovoljno deskriptivna imena
- koristiti donju crtu (eng. *undescore*) kako bi se poboljšala čitljivost i razumljivost koda. Primer:
```
get_best_fit_model();
int x_coordinate;
```
- po potrebi koristiti duža imena kako bi se poboljšala čitljivost i razumljivost koda.
- ne treba koristiti veličinu slova za razlikovanje dva imena. Sva imena treba da budu jedinstvena, bez obzira na veličinu slova koja koriste. Loš primer pisanja imena:
```
int myvar;
int MyVar;
```

Neka standardna imena za različite kodne elemente data su u nastavku. Iako jasna, i dalje se preporučuje davanje dužih imena u datim slučajevima.

Example: standard short names

c	characters
i, j, k	indices
n	counters
p, q	pointers
s	strings

Example: standard suffixes for variables

_ptr	pointer
_file	variable of type file*
_fd	file descriptor

4.6.1. Imena promenljivih

Prilikom pisanja imena potrebno je voditi računa o tome da se ne prave *skrivene promenljive*. Primer:

Example: hidden variable

```
int total;
int func1(void)
{
    float total;       /* this is a hidden variable */
    ...
}
```

Example: no hidden variable

```
int total;
int func1(void)
{
    float grand_total; /* internal variable is unique */
    ...
}
```

4.6.2. Kapitalizacija

Sledeća pravila pisanja velikih i malih slova su preporučljiva, budući da povećavaju čitljivost i razumljivost samog koda.

- **Promenljive**: reči se pišu malim slovima, razdvajanje se vrši pomoću donje crte (_). Primer: `my_new_variable`
- **Imena funkcija**: prvo slovo svake reči se piše velikim slovom, ne koristi se donja crta za razdvajanje. Primer: `MyFunction`
- **Konstante**: svaka reč unutar konstante se piše velikim slovima, razdvajanje se vrši pomoću donje crte (_). Primer: `MAX_COUNT`

5. Tipovi podataka i izrazi

5.1. Promenljive

Prilikom deklaracije promenljivih istog tipa, promenljive deklarisati u posebnim linijama, osim u slučaju kada promenljive same sebe objašnjavaju i kada su međusobno slične.

```
int year, month, day;
```

Ukoliko je prilikom deklaracije promenljive potrebno dodati kratak komentar koji je objašnjava, on se radi u istom redu

```
int x; // komentar
int y; // komentar
```

5.2. #define direktiva

Prilikom definisanja konstanti preporučljivo je koristiti `#define` direktivu. Ona povećava čitljivost koda, ali i omogućava lak mehanizam izmene konstantne vrednosti uniformno u kodu. Primer definisanja konstante pomoću direktive `#define`.

```
#define MAX_VALUE 1000
```

Prilikom definisanja konstanti pomoću `#define` direktive, potrebno je imati na umu da se ime konstante u kodu prilikom preprocesiranja zameni sa navedenim izrazom koji se nalazi sa desne strane. Kako se nekad vrednost konstante definiše pomoću izraza, potrebno je taj izraz smestiti unutar zagrada, čime se izbegavaju mogućnosti grešaka prilikom upotrebe konstante.

```
#define MAX_VALUE (500 + 500)
```

5.3. Enumerisani tipovi

Koriste se za asocijaciju imena konstanti i njihovih vrednosti. Predstavljaju alternativan način za definisanje konstantnih vrednosti. Prilikom definisanja enumerisanih tipova, potrebno je koristiti indentaciju i poravnanja, a identifikatore unutar enumerisanih tipova potrebno je pisati u novom redu. Slede dva načina definisanja enumerisanih vrednosti.

```
enum position
{
    LOW,     //implicitno 0
    MEDIUM, //implicitno 1
    HIGH     //implicitno 2
}

enum error_code
{
    OK = 0,
    GOOD = 0,
    BAD = -1,
    FATAL = -2
}
```

5.4. Brojevi

Prilikom navođenja brojčanih vrednosti promenljivih potrebno je koristiti sledeća pravila.

- Realni brojevi moraju imati bar po jednu cifru i sa desne i sa leve strane decimalne tačke
  `0.5  5.0  1.0e+33`
- Heksadecimalne vrednosti započeti sa *0x* (nulom i malim slovom *x*) koje slede cifre i velika slova A-F.
  `0x123  0x1AF2`
- Duge konstante se završvaju velikim slovom L
  `123L`

5.5. Konverzije tipova

Prilikom konverzije tipova, uvek je preporučljivo koristiti eksplicitnu konverziju. U nastavku su data dva različita tipa konverzije: implicitno i eksplicitno.

```
// eksplicitna konverzija - preporuceno
float f;
int i;
...
f = (int) i;
```

```
// implicitna konverzija
float f;
int i;
...
f = i;
```

5.6. Pokazivači

Prilikom definisanja pokazivača, potrebno je simbol * postaviti uz ime promenljive, a ne uz tip promenljive. Ovim se izbegava greška prilikom definisanja više pokazivača istog tipa u jednom redu, kada se * odnosi samo na prvu promenljivu, tj. samo na prvo ime. Na ovaj način bi se definisao samo jedan pokazivač, dok ostale promenljive ne bi bili pokazivači.

```
Ispravno: int *a, *b, *c;
Pogrešno: int* a, b, c;
```

6. Naredbe i kontrola toka izvršavanja

6.1. Pozicija naredbi

Svaku naredbu je potrebno pisati u pojedinačnom redu, osim u slučaju pisanja *for* petlji. Takođe, potrebno je izbegavati naredbe na koje utiče redosled izvršavanja operacija. Na primer, promenljive sa unarnim operatorom ++ ili --, potrebno je pisati u pojedinačnom redu.

6.2. Zagrade

Spoj naredbi, koje se još naziva blok naredbi, se sastoji od liste naredbi smeštenih unutar zagrada. Stil pisanja zagrada koji se preporučuje je *Braces-Stand-Alone* stil, gde se zagrade smeštaju u samostalnu liniju, tako da su poravnane po kolonama spram nivoa na kom se nalaze.

```
if (a > b)
{
    max = a;
    min = b;
}
```

Iako jezik C ne zahteva upotrebu zagrada oko pojedinačnih naredbi, postoje situacije u kojima zagrade omogućavaju veću preglednost koda. Ugnježdena uslovna izvršavanja ili petlje često postaju značajno jasniji dodavanjem zagrada. Ovaj problem je primetan, kada su uslovi

izvršavanja izrazito dugački i kompleksni, čime skreću pažnju sa samostalne naredbe, koju iz ovog razloga treba smestiti unutar zagrada.

```
for (dp = &values[0]; dp < top_value; dp++)
{
    if (dp->d_value == arg_value
     && (dp->d_flag & arg_flag) != 0)
    {
        return (dp);
    }
}
return (NULL);
```

```
for (dp = &values[0]; dp < top_value; dp++)
    if  (dp->d_value == arg_value &&
        (dp->d_flag & arg_flag) != 0)
        return (dp);
return (NULL);
```

Ukoliko je neki blok naredbi dugačak (duži od 40 linija) ili sadrži nekoliko ugnježdenih blokova, kraj bloka možemo označiti komentarom u liniji zatvorene zagrade tog bloka.

```
for (sy = sytable; sy != NULL; sy = sy->sy_link)
{
    if (sy->sy_flag == DEFINED)
    {
        ...
    }           /* if defined       */
    else
    {
        ...
    }           /* if undefined     */
}               /* for all symbols  */
```

Ukoliko *for* ili *while* petlja nema telo, tj. čitava funkcionalnost je implementirana unutar parametara petlje, simol *;* je potrebno postaviti u sledećem redu. Takođe, poželjno je staviti komentar koji označava da je u petlja bez tela namerna.

6.3. Naredbe kontrole selekcije toka izvršavanja koda

6.3.1. IF naredba

Ukoliko postoji jedna naredba unutar *if* naredbe, potrebno je izvršiti njenu indentaciju u odnosu na *if* naredbu. U ovom slučaju,nije potrebno pisati vitičaste zagrade.
Primer:

```
if (expression)
    one_statement;
```

Ukoliko postoji blok naredbi unutar *if* naredbe, potrebno je pisati vitičaste zagrade, uz obaveznu indentaciju.

```
if (expression)
{
    statement_1;
    ...
    statement_n;
}
```

6.3.2. IF-ELSE naredba

Slično, ukoliko postoji jedna naredba unutar *if* i *else* sekcije, obavezna je indentacija. U ovom slučaju takođe, nije potrebno pisati vitičaste zagrade.

```
if (expression)
    one_statement;
else
    else_statement;
```

Moguće je i da jedna od sekcija (*if* ili *else*) sadrži blok naredbi. U tom slučaju, datu sekciju potrebno je pisati unutar vitičastih zagrada, uz obaveznu indentaciju.

```
if (expression)
    one_statement;
else
{
    statement_1;
    …
    statement_n;
}
```

6.3.3. ELSE-IF konstrukcija

Preporučuje se sledeći način pisanja *else-if* konstrukcije.

```
if (expression)
    one_statement;
else if (expression2)
    else_if_statement;
else
    else_statement;
```

6.3.4. IF-IF-ELSE konstrukcija

Budući da je *else* deo opcion, njegovo izostavljanje može rezultovati u dvosmislenosti. Stoga, potrebno je koristiti vitičaste zagrade, kako bi se to izbeglo. U nastavku su data dva primera koda, prvi - koji daje očekivan ishod, i drugi - koji ne daje očekivan ishod iz gore navedenih razloga.

Example: braces produce desired result

```
if (n > 0)
{
    for (i = 0; i < n; i++)
    {
        if (s[i] > 0)
        {
            printf("...");
            return(i);
        }
    }
}
else    /* CORRECT -- braces force proper association  */
    printf("error - n is zero\n");
```

Example: absence of braces produces undesired result

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0)
        {
            printf("...");
            return(i);
        }
else    /* WRONG -- the compiler will match to closest */
        /* else-less if */
    printf("error - n is zero\n");
```

6.3.5. Switch naredba

Radi čitljivosti i razumljivosti koda, *switch* naredba treba da ima sledeći oblik:

```
switch (expression)
{
    case aaa:
        statement[s]
        break;
    case bbb: /* fall through */
    case ccc:
        statement[s]
        break;
    default:
        statement[s]
        break;
}
```

Treba primetiti da, ukoliko postoji namerno propadanje (*fall-through*) unutar *switch* naredbe, isto treba komentarisati. Takođe, svaka *switch* naredba treba da sadrži i *default* deo. *Default* deo se uvek nalazi na kraju, i ne mora da sadrži *break* naredbu, ali se njeno pisanje preporučuje, radi konzistentnosti koda.

6.4. Petlje

6.4.1. *for* petlja

U slučaju da telo petlje sadrži samo jednu naredbu, nije potrebno smestiti naredbu unutar zagrada. Međutim, ukoliko telo petlje sadrži blok naredbi, potrebno ih je ograničiti zagradama.

```
for (expression)
    one_statement;

for (expression)
{
    statement_1;
    statement_2;
    statement_3;
}
```

Prilikom velikog broja parametara *for* petlje, česta je situacija da se *for* petlja ne može smestiti u jednu liniju. Kako bi se rešio ovaj problem, parametri se razdvajaju u tri linije, gde prvu čini inicijalizacija, drugu uslov izvršavanja petlje i treći promene na kraju iteracije.

```
for (curr = *listp, trail = listp;
    curr != NULL;
    trail = &(curr->next), curr = curr->next)
{
    statement_1;
    ...
    statement_n;
}
```

6.4.2. *while* petlja

Za *while* petlju važe ista pravila kao i za *for*, u slučaju da telo petlje čini jedna naredba, nije potrebno koristiti zagrade. U slučaju da telo petlje čini blok naredbi, treba ga smestiti unutar zagrada.

6.4.3. *do while* petlja

U slučaju *do while* petlje koristiti zagrade bez obzira da li telo petlje čini jedna ili više naredbi.

6.5. *goto* i *break* naredba

6.5.1. *goto* naredba

Ukoliko je moguće, *goto* naredbu je preporučljivo izbegavati. Primer slučaja u kom je korisno upotrebiti *goto* naredbu za izlazak iz petlje dat je u nastavku.

```
    for (...)
    {
        for (...)
        {
            ...
            if (disaster)
            {
                goto error;
            }
        }
    }
    ...
error:
    error processing
```

6.5.2. *break* naredba

Naredba *break* se koristi za izlazak iz unutrašnjosti *for*, *while*, *do while* petlje ili *switch* naredbe u proizvoljnom trenutku, nasuprot napuštanju prouzrokovanim nezadovoljavanjem testa petlje.

7. Literatura

NASA C stil kodovanja - https://ntrs.nasa.gov/citations/19950022400