Zapiski s priprav

Funkcije

ACM priprave na tekmovanja

Različica z dne 22. marec 2025

Kazalo

1 Osnovna sintaksa

Ko pišemo daljše programe, se pogosto zgodi, da večkrat uporabimo nek podoben kos kode. Da si v takih situacijah olajšamo življenje, lahko uporabimo *funkcije*. Funkciji podamo nekaj *parametrov* (včasih se uporabi tudi beseda *argumentov*), ona nekaj melje in nam potem *vrne* neko vrednost. Poleg tega, da samo vrne neko vrednost, ima lahko funkcija tudi kakšne stranske učinke. Lahko na primer kaj izpiše z uporabo `printf`, ali pa spremeni vrednost neke spremenljivke. Včasih so parametri ali vrnjena vrednost nepotrebni, zato lahko napišemo tudi funkcije brez parametrov ali funkcije, ki ne vrnejo ničesar.

Najbolj osnovna oblika funkcije je funkcija, ki ne sprejme nobenega parametra in ne vrne ničesar. Če hočemo, da ta funkcija ni neuporabna, bo morala imeti kakšen stranski učinek. V spodnjem primeru funkcija izpiše besedilo.

```c
#include <stdio.h>

void moja_funkcija() {
    // koda, ki se bo izvedla, ko pokličemo funkcijo
    printf("Zdravo\n");
}

int main() {
    moja_funkcija(); // pokličemo funkcijo
    return 0;
}
```

Poglejmo si malo bolj podrobno, kaj posamični deli zgornjega programa naredijo. Pri definiciji funkcije smo prvo napisali `void` — to je posebna oznaka, ki pomeni, da funkcija ne vrne ničesar. Za tem je beseda `moja_funkcija`, ki označuje ime funkcije, ki jo definiramo. Tako kot spremenljivke lahko funkcije poimenujemo, kakor želimo. Za imenom funkcije je običajen oklepaj, ki zaznamuje začetek seznama argumentov. V tem primeru argumentov nimamo, zato seznam takoj končamo z zaklepajem. Na koncu še v zavite oklepaje postavimo kodo, ki jo bo funkcija izvedla.

Če želimo funkciji podati argumente, jih zapišemo kot spremenljivke med oklepaje v definiciji funkcije:

```c
#include <stdio.h>

void plusi(int n) {
    // izpišemo n plusov
    for (int i = 0; i < n; i++) {
        printf("+");
```

```
    }
    printf("\n"); // končamo z znakom za novo vrstico
}

int main() {
    plusi(12); // pokličemo funkcijo s parametrom 12
    return 0;
}
```

Zgoraj smo v oklepaje za imenom spremenljivke postavili `int n`. Beseda `int` pove, da je ta parameter celo število, `n` pa je ime za parameter. Ime, ki smo si ga izbrali za parameter, uporabljamo v telesu funkcije, da dostopamo do vrednosti v parametru.

Če je naš parameter seznam, to napišemo kot spodaj.

```
#include <stdio.h>

void izpisi_prve_tri(int seznam[]) {
    printf("%d %d %d\n", seznam[0], seznam[1], seznam[2]);
}

int main() {
    int stevila[] = {7, 8, 6, 1, 2, 6, 3};
    izpisi_prve_tri(stevila); // izpiše "7 8 6"
    return 0;
}
```

Da funkcija vrne neko vrednost, moramo namesto `void` napisati tip vrednosti, ki jo vrne, nato pa nekje v notranjosti funkcije uporabiti `return`. Poglejmo si enostaven primer, kjer s funkcijo izračunamo vsoto dveh števil.

```
#include <stdio.h>

int vsota(int a, int b) {
    return a + b;
}

int main() {
    int rezultat = vsota(3, 4);
    printf("%d\n", rezultat);
    return 0;
}
```

Kot smo navajeni, se ukazi v telesu funkcije izvajajo po vrsti od zgoraj navzdol. V njej lahko uporabimo vse, kar smo se do zdaj naučili (lahko ustvarjamo nove spremenljivke, uporabimo pogojne stavke, zanke, …). Vredno omembe je, da izvajanje funkcije konča takoj, ko se izvede prvi `return`, tudi če je za tem še nekaj kode.

V spodnjem bloku kode funkcije vrne logično vrednost `bool`, ki je ena od `true` ali `false`. Funkcija deluje pravilno, saj če bo `crka` enaka `'a'`, `'e'`, `'i'`, `'o'` ali `'u'`, bo funkcija končala znotraj pogojnega stavka, sicer pa bo nadaljevala do `return false` na koncu.

```c
#include <stdio.h>

bool je_samoglasnik(char crka) {
    if (crka == 'a' || crka == 'e'
        || crka == 'i' || crka == 'o'
        || crka == 'u')
    {
        return true;
    }
    return false;
}

int main() {
    char c;
    scanf("%c", &c); // prebere eno črko

    if (je_samoglasnik(c)) {
        printf("To je samoglasnik!\n");
    } else {
        printf("To pa ni samoglasnik.\n");
    }
    return 0;
}
```

2 Spremenljivke in funkcije

Spomnimo se, da lahko spremenljivke definiramo na dva načina. Tistim, ki so definirane na vrhu, izven funkcij, pravimo *globalne spremenljivke*, tistim, ki so pa definirane v neki funkciji, pa *lokalne spremenljivke*. Globalne spremenljivke so vidne povsod, torej v vseh

funkcijah, medtem ko lahko lokalne spremenljivke uporabljamo samo v funkciji, v kateri smo jih definirali.

```c
#include <stdio.h>

// tu so globalne spremenljivke
int g = 12;

void funkcija() {
    // v tej funkciji lahko dostopamo do g
    printf("%d\n", g);
    g += 3;
}

int main() {
    int n = 200;
    funkcija();
    // tu lahko dostopamo do g in n
    printf("%d, %d\n", g, n);
    return 0;
}
```

Seveda, če globalno spremenljivko nekje spremenimo, se bo spremenila tudi za vse druge funkcije. V zgornjem primeru `funkcija` najprej izpiše `g` (12) in ga nato poveča na 15. Ko se `g` izpiše drugič, je zato enak 15.

Če funkciji podamo neko spremenljivko kot parameter in poskušamo to spremenljivko v funkciji spreminjati, se obnaša na dva različna načina, odvisno od tega, ali je spremenljivka seznam ali ne.

Če spremenljivka ni seznam (npr. `int` ali `char`) se za funkcijo ustvari kopija te spremenljivke. Če jo v funkciji spremenimo, to ne bo spremenilo izvirne spremenljivke, saj v funkciji delamo s kopijo. Če pa argument je seznam, bodo spremembe na seznam v funkciji vplivale na izvirni seznam, saj se ta ne bo prekopiral. V spodnjem primeru sta prikazani obe možnosti.

```c
#include <stdio.h>

void funkcija(int stevilo, int seznam[]) {
    stevilo = 123; // ta sprememba ne vpliva na a v main-u
    seznam[0] = 123; // ta sprememba vlpliva na sez v main-u
}

int main() {
```

```c
    int a = 10;
    int sez[] = {10};
    funkcija(a, sez);
    // a bo še vedno 10, medtem, ko bo seznam enak {123}
    return 0;
}
```

Sprva ne vidimo smisla, za drugačno obravnavo seznamov, a se izkaže, da je lahko ta lastnost tudi uporabna. Če bi želeli napisati funkcijo, ki sprejeme seznam in njegovo dolžino, potem pa obrne vrstni red elementov tega seznama, lahko to napišemo takole:

```c
#include <stdio.h>

void obrni(int seznam[], int dolzina) {
    // Obrnemo vrstni red prvih dolzina elementov seznama
    for (int i = 0; i < dolzina / 2; i++) {
        int vmesna = seznam[i];
        seznam[i] = seznam[dolzina-i-1];
        seznam[dolzina-i-1] = vmesna;
    }
}

int main() {
    int n;
    int seznam[1000];

    // preberemo seznam dolžine n
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &seznam[i]);
    }

    obrni(seznam, n);

    // izpišemo obrnjen seznam
    for (int i = 0; i < n; i++) {
        printf("%d\n", seznam[i]);
    }

    return 0;
}
```

3 Rekurzija

Rekurzija je način pisanja programa na način, da funkcija kliče samo sebe. Pogosto lahko zanko nadomestimo z rekurzijo, če nam je tak način programiranja ljubši. Recimo, da želimo sešteti vsa števila med 1 in nekim n. Z zanko bi to naredili tako, da si pripravimo spremenljivko `vsota`, potem pa vanjo prištevamo števila, ki jih vidimo v zanki. Vsoto prvih n števil pa izračuna tudi naslednja funkcija:

```
int vsota(int n) {
    // osnovni pogoj
    if (n <= 0) {
        return 0;
    }

    // korak
    return n + vsota(n - 1);
}
```

Najprej napišemo *osnovni pogoj*, ki v primeru, da je argument `n` manjši ali enak 0, takoj vrne 0. Ta pogoj nujno potrebujemo, saj se bo sicer rekurzija ponavljala v neskončnost. Po osnovnem pogoju zapišemo *korak rekurzije*, ki izračuna `vsota(n)` s pomočjo vrednosti funkcije v nekem manjšem številu, v našem primeru kar `vsota(n - 1)`.

Poglejmo si, kaj se zgodi, če pokličemo `vsota(4)`. Funkcija izračuna $4 + \text{vsota}(3)$, ampak da bi izračunali to, moramo prvo izračunati `vsota(3)`. Pri računanju `vsota(3)` izračunamo $3 + \text{vsota}(2)$, za kar moramo izračunati `vsota(2)`. Potem podobno nadaljujejo, pri računanju `vsota(2)` računamo `vsota(1)`, pri računanju tega pa izračunamo `vsota(0)`. Tu se rekurzija ustavi, saj pridemo v prvi pogojni stavek, in takoj vrnemo vrednost 0. Potem lahko izračunamo $\text{vsota}(1) = 1 + 0$, iz česar dobimo $\text{vsota}(2) = 2 + 1$, potem $\text{vsota}(3) = 3 + 2$, $\text{vsota}(4) = 4 + 6$ in na koncu $\text{vsota}(5) = 5 + 10 = 15$.