C++

Funkcije

Uvod

- **sastavni dio C++ programa**
- **tipični profesionalni program sastoji se od više funkcija imenovanih prema zadacima koje obavljaju**
- **odvojene zadatke treba stavljati u odvojene funkcije i "pozivati" kad je potrebno – prednosti:**
  - dobro napisana funkcija može biti spremljena i korištena u drugim programima
  - razbijanje koda u funkcije doprinosi modularnosti programa. Kôd postaje čitljiviji i razumljiviji.

Podjele

- Specifikacija se odnosi na ulaz i izlaz iz funkcije (važno kod rada sa funkcijama iz biblioteke)
- Dizajn se odnosi na zadatak funkcije (važan je kod razvoja – pisanja koda vlastitih funkcija)

- kod poziva funkcije podaci ulaze i izlaze iz funkcije
- funkcije se prema nastanku mogu podijeliti u dvije grupe:
  - **funkcije iz biblioteke**
  - **funkcije koje definiraju programeri**
- prema prijenosu podataka mogu se funkcije podijeliti na:
  - **funkcije koje ne vraćaju vrijednost**
  - **funkcije koje vraćaju jednu vrijednost**
  - **funkcije sa “pass by reference” koje mijenjaju vrijednosti više u/i podataka**

Upotreba funkcija u programu – temeljni pojmovi

- **Deklaracija funkcije** (prototype) – slično deklaraciji varijable
- **Poziv funkcije** (function call)
- **Definicija funkcije** (function definition)
- **Tip funkcije** (function type)
- **Argumenti** (parametri**) funkcije** (zadaju se prilikom poziva)
- **Povratna vrijednost** (return value, rezultat funkcije)

Funkcije se pišu **kao zasebni blokovi naredbi** prije ili poslije glavne funkcije. Poziv funkcije se obavlja tako da se izvođenje djela kôda trenutno prekine te se prenese u funkciju. Nakon što se kôd u funkciji izvede, program se nastavlja od slijedeće naredbe iza poziva funkcije.

Deklaracija funkcije (function prototype)

- pokazuje da funkcija određenog imena postoji i pokazuje druge informacije o funkciji
- kao i varijable, funkcije moraju biti deklarirane (ili definirane) prije njihove prve upotrebe u programu
- obznanjuje naziv funkcije, broj, redoslijed i tip parametara (formalnih) te tip njene povratne vrijednosti
- deklaracija funkcije može biti izvan svih ili unutar tijela neke od funkcija
- primjer prototipa (deklaracije) funkcije:

```
double kvadrat (float a);
```

Definicija funkcije

- sastoji se od linije zaglavlja (header, po obliku odgovara deklaraciji funkcije) koju slijedi tijelo funkcije ograničeno vitičastim zagradama. Tijelo funkcije sastavljeno je od naredbi koje se u funkciji izvršavaju.
- sadrži opis što i kako funkcija radi
- <u>mora biti smještena izvan tijela drugih funkcija</u>
- primjer:

```
double kvadrat (float a)
//definicija funkcije sadrži i tijelo funkcije
    {return a * a;}
```

Poziv funkcije

- naredba koja uzrokuje prijenos upravljanja od jedne funkcije drugoj funkciji i može prenijeti vrijednosti podataka funkciji
- u pozivu se navodi naziv funkcije sa parametrima u zagradi
- broj argumenata u pozivu funkcije (stvarni) mora biti jednak broju argumenata u definiciji funkcije (formalni), a moraju odgovarati i tipovi
- poziv funkcije može biti dio naredbe, aritmetičkog izraza ili argument poziva funkcije
- primjeri:
  - `cout << kvadrat(3)<<endl;`
  - `h=kvadrat(a)+kvadrat(b);`

Lista argumenata funkcije

- argumenti unutar zagrada iza imena funkcije su podaci koji se predaju funkciji prilikom njena poziva
- broj i tip argumenata u zagradi može biti proizvoljan (ili 0)
- argument u definiciji je **formalni argument** (simboličko ime)
- pri pozivu funkcije formalni argument inicijalizira se **stvarnim argumentom** (poprimi njegovu vrijednost)
- vrijednost koju funkcija vraća, ime, broj argumenata, njihov redoslijed i tip nazivaju se **potpisom funkcije**

Primjer – kvadrat - nastavak

```
#include <iostream>
#include <iomanip>
using namespace std;

double kvadrat (float);

int main() {
   for(int i=1; i<=10; i++)
         cout << setw(5)<<i<<setw(10)<<kvadrat(i)<<endl;
return 0;
}

double kvadrat(float x) {
   return x*x;}
```

Tip funkcije

- tip ispred imena funkcije obavezan je, a određuje kakvog će tipa biti podatak kojeg će funkcija vraćati pozivatelju kao rezultat svog izvođenja
- tip se funkciji dodjeljuje prilikom deklaracije tako da se ispred imena funkcije navede identifikator tipa
- konkretna vrijednost koju funkcija vraća određuje se pomoću naredbe `return` u definiciji funkcije (tip rezultata mora odgovarati tipu funkcije ili se obavlja pretvorba)
- ako funkcija ne treba vratiti vrijednost (npr. kad samo ispisuje) ona se deklarira tipom `void`, naredba `return` tada ne sadrži nikakav podatak, a može se i izostaviti
- funkcije tipa `void` stoje zasebno u naredbenom retku – ne mogu biti s desne strane znaka pridruživanja

Povratak i vraćanje vrijednosti iz funkcije

- povratak je moguć s bilo kojeg mjesta unutar funkcije – vrši se bezuvjetni skok na drugu lokaciju
- naredba `return` može se pojavljivati i na više mjesta u tijelu funkcije
- za vraćanje vrijednosti iz funkcije potrebni su:
  - odgovarajući tip funkcije
  - naredba `return` u tijelu funkcije (sa vrijednošću varijable ili izraza - može biti u zagradama)
- funkcije tipa `void` ne vraćaju vrijednosti, a za povratak mogu koristiti samo `return;`
- `return(0)` za povratak iz funkcije `main()` nije obavezno koristiti - vraćanje nule OS-u se uglavnom podrazumijeva

Prijenos argumenata po vrijednosti

- formalni argument i vrijednost koja se prenosi nisu međusobno povezani – **nalaze se u dva različita područja u memoriji**
- funkcija napravi ili dobija kopiju vrijednosti svakog argumenta i cijelo vrijeme radi s kopijom – **prijenos po vrijednosti (pass by value ili call by value)**
- kada se funkciji argumenti prenose po vrijednosti, njihova se prvotna vrijednost ne mijenja

Prijenos argumenata po referenci

- za **vraćanje dviju (ili više) vrijednosti**
- u listi argumenata su vrijednosti (varijable) sa kojima **funkcija radi direktno** (ne sa njihovim kopijama)
- **simbol reference &** pokazuje argumente kojima će funkcija pri pozivu promijeniti vrijednosti
- korištenjem referenci na taj način **kreiraju se nova (druga) imena ili *aliases* za originalna imena varijabli** – za nova imena nije potrebno osigurati memoriju, ona su samo reference

Primjer – kvadar

```
#include <iostream>
using namespace std;

void kvadar_vol_oplos (double, double, double, double&, double&);

int main() {
    double a, v;
    double x=6.3, y=7.2, z=1.5;
    kvadar_vol_oplos (x, y, z, a, v);
    cout << "oplosje = "<<a<<"volumen = "<<v<<endl;
return 0;}

void kvadar_vol_oplos (double s1, double s2, double s3, double& op, double& vol)
{   op = 2*s1*s2 + 2*s2*s3 + 2*s1*s3; vol = s1*s2*s3; }
```

Područje dosega - scope

- **Područje dosega (scope) određeno je mjestom deklaracije identifikatora,** pa C++ razlikuje:
  - **file scope** – varijable deklarirane izvan i prije tijela svih funkcija – dostupne svim funkcijama te datoteke izvornog koda – na jednoj memorijskoj lokaciji (ne preporuča se, bolje je doseg širiti kroz listu argumenata) - globalno
  - **function scope** – varijable deklarirane u funkcijama – vrijede samo u funkcijama u kojima su deklarirane – nalaze se na različitim memorijskim lokacijama - lokalno
  - **block scope** – identifikator deklariran između { i }
- područje dosega može se proširiti i na različite datoteke

Memorijske klase

- C++ dozvoljava modificiranje pravila za područje dosega i memoriju isticanjem memorijske klase varijable u njenoj deklaraciji
- memorijska klasa varijable odnosi se na područje dosega (scope) varijable i dužinu vremena (tijekom izvođenja programa) za koje je memorija rezervirana za varijablu
- specifikatori memorijskih klasa u C++:
  - register
  - auto
  - static
  - extern

Klasa “register”

- želju da se vrijednost varijable spremi u registre ističemo pomoću ključne riječi register u deklaraciji

```
register double x;
```

- može učiniti bržim pristup vrijednosti i tako poboljšati performanse programa
- današnji kompajleri dovoljno su sofisticirani da sami identificiraju često korištene varijable pa to rade automatski

Klasa “auto”

- Varijable deklarirane u funkcijama su “by default” (podrazumijevano) auto, što znači da memoriju oslobađaju nakon završetka izvođenja funkcije, pa se riječ auto rijetko koristi

Klasa “static”

- Memorija se rezervira i inicijalizira (može u deklaraciji) prvi put kad je funkcija pozvana, ali se vrijednost varijable u memoriji zadrži i nakon što je funkcija završila s izvođenjem (suprotno od auto) i tako prenosi slijedećem izvođenju funkcije, npr.

```
static double max=-1e50;
```

- Kada nije inicijalizirana u deklaraciji, C++ inicijalizira static varijable na nulu.
- Možemo ju koristiti i kada želimo biti sigurni da globalna varijabla/funkcija u jednoj datoteci nije slučajno deklarirana kao extern i korištena u drugoj datoteci – ograničavanje varijable/funkcije na korištenje samo jednoj datoteci

Klasa “extern”

- C++ zahtjeva upotrebu oznake `extern` memorijske klase za globalne varijable u slučaju kad je program “razbijen” u više datoteka
- U jednoj je datoteci globalna varijabla deklarirana bez oznake memorijske klase, a u drugim datotekama, ključna riječ `extern` koristi se prije tipa varijable, npr.

```
extern type variable;
```

- Deklaracija sa `extern` određuje samo ime i tip varijable, ne rezervira memoriju za varijablu, omogućuje samo korištenje te varijable
- **Funkcije ne moraju biti deklarirane sa** extern

Funkcije sa podrazumijevanim argumentima

- Podrazumijevani (default) argument je argument kojem se pridružuje određena vrijednost u slučajevima kad je izostavljen iz funkcijskog poziva (odstupanje od NOT pravila)
- Uobičajeno je koristiti podrazumijevani argument ako on <u>obično (ne i uvijek) ima istu vrijednost</u> ili kad je funkcija promijenjena u odnosu na svoju izvornu definiciju (kako bi se izbjeglo mijenjanje svih poziva funkcije)

Primjeri funkcija sa podrazumijevanim vrijednostima

```
void com_time (double, double=25, int=5);
ili
void com_time (double, double d=25, int nl=5);
...
com_time (40);
com_time (30, 20);
com_time (35, 30, 8);
...
void com_time (double v, double d, int nl)
...
```

Opis

- Podrazumijevane vrijednosti argumenta moraju biti određene u deklaraciji funkcije
- C++ ima strogo pravilo o tome koji argumenti mogu biti odabrani kao proizvoljni: **u listi argumenata "obični" (nepodrazumijevani) argumenti ne mogu biti pisani nakon bilo kojeg podrazumijevanog** (zbog načina pozivanja funkcije)
- Možemo koristiti podrazumijevane vrijednosti i kada imamo funkciju za koju vrijednosti prenosimo "by reference", npr.

```
void com_time (double&, double& = 25, int& = 5);
```

Poziv funkcija sa podrazumijevanim argumentima

- kod poziva funkcije, može se koristiti **manje od ukupnog broja** argumenata u listi
- funkcija mora biti pozvana najmanje sa svim “običnim” (nepodrazumijevanim) argumentima
- ako funkciju iz primjera pozivamo s jednim ili dva argumenta, pri izvođenju se koriste sve ili neke podrazumijevane vrijednosti
- Nema načina da se koristi podrazumijevana vrijednost za drugi argument, a ne koristi za treći

Preopterećenje funkcija

- **definiranje dviju ili više funkcija sa istim imenom (overloading, preopterećenje)**
- takve funkcije tipično izvode slične zadatke iako su njihova imena identična (npr. ako sa različitim brojem argumenata (ili sa različitim tipovima podataka) želimo izvesti sličan zadatak, recimo rotaciju – riješiti!)
- programer jednostavnije koristi funkcije
- funkcije istog imena moraju imati deklaracije koje se jasno razlikuju i biti istog tipa (npr. void) kako ne bi došlo do zabune ili pogreške, paziti treba i kod automatske konverzije tipa podataka

Usporedba globalnih i static varijabli

- za vrijeme izvođenja kreira se i traje tzv. permanentna memorija; čuva naredbe, globalne i static varijable; tzv. heap koji svoju veličinu ne mijenja čitavo vrijeme izvođenja, a nalazi se na nižim memorijskim adresama
- stack dio memorije (više memorijske adrese) povećava se i smanjuje kako se funkcije pozivaju i izvode - memorija se rezervira i oslobađa za auto varijable funkcija
- raspoloživa je memorija između dijelova heap i stack
- static varijablama može se pristupiti samo iz funkcije u kojoj su deklarirane, globalnim se varijablama može pristupiti iz bilo koje funkcije, a ako se koristi extern iz bilo koje datoteke
- bolje je koristiti static varijable: povećana je strukturiranost programa i limitiranost pristupa varijablama

Programi u više datoteka

- Izvorni kod velikih programa često se “razbija” u više odvojenih datoteka i kompilira odvojeno jer se sa malim datotekama obično radi jednostavnije i efikasnije, a poboljšava se proces razvoja programa
- Da bi se dobilo izvršnu datoteku, povezuje se sve objektne kodove gotovih i onog dijela programa koji je u izradi.
- Prednosti:
  - vrijeme kompilacije je smanjeno,
  - više programera može raditi na istom programu,
  - manja je vjerojatnost slučajne promjene ispravnog dijela programa

Deklaracije funkcija u datotekama zaglavlja

- Kad se izvorni kod razbija u više modula
- Deklaracije funkcija stavljaju se obično u zasebne datoteke zaglavlja (koje u principu ne sadrže i definicije funkcija)
- Imena datoteka zaglavlja nalaze se u dvostrukim navodnicima kada ih se prvo traži u tekućem folderu
- Ključna riječ extern ispred deklaracije označava da su funkcije definirane u nekoj drugoj datoteci
- Vrlo je važna dobra organizacija datoteka i koda prilikom razvoja složenih programa