Objektno orjentisano programiranje

Uvod u objektno orjentisano programiranje (OOP) Koncepti OOP. Razlike C i C++.

Softver – Obični korisnik

- Porast procesorske moći računara i pojeftinjenje komponenti omogućili su da "niskobudžetni" korisnici mogu da priušte sebi računare za posao i zabavu.
- Softveri za "obične" korisnike morali su da budu pisani na osnovu strategije odbrambenog programiranja (trebalo je predvidjeti korektivnu akciju za svaku glupost koji ovi korisnici pokušaju) i morali su da se izvršavaju interaktivno (na osnovu nekih intuitivnih predstava) jer se ovi korisnici nijesu mogli natjerati da pamte naredbe ili složene procedure (ili nijesu htjeli da plate za takve softvere).
- Da stvar bude gora, obični korisnici su kada su shvatili moć računarskih sistema počeli da postavljaju sve veće i veće zahtjeve.

Softver – Proizvođač softvera

- Da bi proizvođač softvera zaradio mora da izađe u susret zahtjevima korisnika (u suprotnom, preoteće mu ih drugi proizvođač).
- To znači da su softverske firme morale da brzo proizvode nove verzije softvera.
- Brzo iz dva razloga: da bi što prije izašle u susret korisničkim zahtjevima i da bi što manje platile programere (na zapadu programeri rade po satnici).
- Kako su softveri bivali sve komplikovaniji i komplikovaniji to su nove verzije morale polaziti od koda stare verzije i na osnovu prepravki se dolazilo do nove verzije.

Softver - Sadržaj

- Programski moduli koji su sačinjavali softvere su prevashodno bile funkcije.
- Podaci nad kojima su funkcije operisale bili su po pravilu strukture (samo struktura može da predstavi složeniji podatak: radnika, studenta, prozor u grafičkom okruženju, grafik itd.).
- Svaka funkcija koja je radila sa strukturama morala je da zna na koji je način struktura napravljena (od čega se sastoji).
- E tu leži zec...

Softver - Prepravke

- Prepravke softvera su bile komforne dok se prepravljaju funkcije, ali kada se prepravi jedna struktura, to je značilo da se moraju prepraviti sve funkcije koje komuniciraju sa tom strukturom, tj. koriste je.
- Svaka prepravka funkcije se morala testirati.
- Najveći problem je to što prepravka strukture dovodi do nelokalizovanih prepravki programa, odnosno najvjerovatnije indukuje prepravku kompletnog programa!
- Jedna prepravka indukuje mnoštvo drugih prepravki, a svaka prepravka znači mnogo testiranja.
- Programeri su svoje korisno vrijeme trošili, ne na pisanje programa, već na njihovo testiranje!!!

Softverska kriza

- Da bi zaradile, softverske kuće su na tržište iznosile programe koji nijesu bili dovoljno testirani i ti programi su imali mnogo grešaka (bugs). Kupci nijesu željeli da kupuju takve softvere ili su tražili povraćaj uloženog novca.
- Mnoštvo softverskih firmi je propalo.
- Postalo je jasno da uskoro nijedna firma neće moći da proizvede iole ispravan softver za dato vrijeme.
- Ovaj događaj se naziva **softverskom krizom**.
- Krajem 80-tih jedna grupa programera na čelu sa Bjarne Stroustrupom dobila je zadatak da prepravi softver za telefonsku centralu.

Rješenje softverske krize

- Grupa je radila za AT&T Bell.
- Stroustrup je uočio da zadatak prepravke softvera **(pisan u C-u, kao i 90% sistemskog softvera tada)** nije moguće obaviti u datom vremenu, na datom kompajleru i sa datim ljudstvom.
- Stoga je potražio nove koncepte na kojima bi se softver mogao realizovati.
- Tada su već postojali **(više od 10 godina)** **objektno-orjentisani (OO) jezici** **(primjer je Smalltalk)**, ali nijesu bili popularni zbog složenosti.
- Stroustrup je došao na spasonosnu ideju: nakalemiti OO koncepte na postojeći (C) kompajler. Dobio je kombinaciju jednostavnosti i moći alatke kojom je riješio svoj problem **(pišući novi kompajler)**, ali i čitavu softversku krizu.

OOP (programiranje) - Koncepti

- Stroustrup je postigao da OOP nije zamjena za strukturno programiranje, već njegovo proširenje. Tokom vremena ćemo naučiti da je ovo proširenje ogromno i da zasjenjuje mnogo toga što smo do sada učili. Ovdje ćemo pomenuti samo osnovne koncepte OOP koje ćemo na primjeru programskog jezika C++ učiti do kraja kursa.
- **Enkapsulacija.** Ovo je najvažniji OO koncept. Ovaj koncept omogućava da se način realizacije djelova koda (zapravo, korisničkih tipova podataka) sakrije od korisnika (u **kapsulu** koja se naziva klasa). Prepravke unutar klase nemaju uticaj na ostatak koda, jer ostatak koda nije nikada znao kako je klasa bila realizovana. Na ovaj način, prepravljanje koda je lokalizovano, čime se podržava reusability (ponovno korišćenje) programa.

OOP - koncepti

- Ponekad se kaže da su u programiranju najvažnije tri stvari: reusability, reusability i reusability, odnosno mogućnost ponovnog korišćenja koda.

- **Apstrakcija.** Ovo je prvi koncept po svojoj logici. Apstrakcija je modelovanje dijela problema razmatranjem odgovarajućeg tipa podataka. Npr. ako je program vezan za rad studentske službe, Student je dio problema i taj problem treba riješiti modelovanjem Studenta. Taj model se kreira kao tip podataka (klasa), odnosno tip podataka nam je dio rješenja problema. Ovo je jedan od elemenata OOP – veliki dio problema se rješava u okviru modela podataka.

OOP Koncepti

- **Nasljeđivanje.** Kreirali smo jednu klasu, recimo -Student. Međutim, sistem posjeduje i podatke tipa Diplomac, koji su različiti unekoliko od ove klase. Jedan način rješavanja problema je da realizujemo dvije klase sa gotovo istim operacijama koje se sa njima mogu vršiti. Ovo je loše, jer ako je bilo koja operacija loše realizovana mi prepravke moramo obaviti na više mjesta u kodu, čime se gube neke dobre strane OOP, a to je lokalizovano prepravljanje grešaka. Umjesto toga, mi možemo reći da klasa Diplomac (naziva se izvedenom klasom) nasljeđuje sve iz klase Student (naziva se osnovnom klasom) uz neke dopune i izmjene.

OOP Koncepti

- **Prijateljstvo.** Više klasa može da radi na rješavanju istog problema. Ponekad postoji potreba da one na neki način razmjenjuju podatke. Da bi se izbjeglo ugrožavanje enkapsulacije **(odnosno situacija da klasa svima dozvoli uvid u svoju realizaciju)** uvodi se koncept prijateljstva, po kome klasa samo određenim klasama (i funkcijama) može dozvoliti pristup svojoj realizaciji.
- **Preklapanje operatora.** OOP i C++ dozvoljavaju da se značenje standardnih operatora predefiniše za klasne tipove podataka. Nema potrebe da naglašavamo koliko je pogodnije i prihvatljivije zapisati sabiranje vektora kao **a+b** nego preko uvođenja funkcije koja bi ovo sabiranje obavila.

OOP Koncepti

- **Polimorfizam.** Ovo je jedan od najsloženijih (i najmoćnijih) koncepata OOP. Jedan dio koda u OOP može da se ponaša različito u zavisnosti od toga na kakve se podatke primjenjuje. Koncept je složen i realizuje se preko: preklapanja funkcija, nasljeđivanja (posebno koncepta virtuelnih funkcija) i preko šablona klasa i funkcija. Koncept ćemo detaljno razmatrati kad za to dođe vrijeme.
- **Izuzeci.** U programiranju, izuzetak je događaj koji se može predvidjeti, a ne može izbjeći (nema papira, djeljenje sa nulom, itd.). OOP i C++ dozvoljavaju obradu izuzetaka tako što definišu mogućnost postojanja programske prečice između mjesta gdje nastaje izuzetak i mjesta gdje se on obrađuje.

Organizacija II dijela kursa

- Nakon ovog kratkog uvoda, prelazimo na izučavanje koncepata koje je C++ dodao na programski jezik C (naročito na verzije prije 1999) , a koji nijesu strogo objektne prirode.
- Nakon toga učimo koncepte OOP u programskom jeziku C++.
- Računske i laboratorijske vježbe prate predavanja.

Razlike C i C++ neobjektne prirode

- Gornji naslov nije baš najpreciziji, jer ćemo uvesti i neke objektne koncepte koje tako nećemo predstaviti. Također, nakon 1999. neke od navedenih razlika su inkorporirane u C.
- C++ posjeduje sve ključne riječi C-a.
- Može se shvatiti kao prošireni ili poboljšani C.
- Svaki ispravno napisan C program može se kompajlirati na svakom C++ kompajleru.
- Prva razlika koju uvodimo je kod deklaracije promjenljivih. C++ doživljava programsku petlju kao ispravan početak bloka naredbi, tako da je dozvoljeno deklarisati promjenljivu u programskoj petlji, npr.:
  for(int i=0;i<100;i++) /*...*/
- Doseg ovako uvedene promjenjive je do kraja tijela for petlje {}.
- **Napomena: Većina C++ kompajlera dozvoljava deklaraciju promjenljive bilo gdje u bloku naredbi prije prve upotrebe promjenljive (ne ekskluzivno na vrhu kao u C-u), ali ne treba zloupotrebljavati ovu mogućnost.**

Razlike kod funkcija

- "Neobjektne" razlike C-a i C++ su najveće u dijelu priče o funkcijama.
- Postoje tri dodatka koja čine funkcije u C++ fleksibilnijim i na neki način boljim od onih u C-u:
  - Inline funkcije (prije C99);
  - Podrazumijevani argumenti
  - Preklopljanje funkcija (function overloading)
- Vidjeli smo da u izvršavanju programa učestvuju: alokacioni zapis **(koji čuva podatke o tekućoj aktivaciji – dijelu koda koji se trenutno izvršava)** i stek **(na kom se nalaze podaci o aktivacijama koje su pozvale aktivni dio koda).**
- Snimanje podataka na stek i preuzimanje podataka sa steka traje neko vrijeme koje može da bude duže od vremena izvršavanja jednostavnih funkcija.

Inline funkcije od C99

- Da bi se izbjeglo gubljenje vremena na izvršavanje kratkih funkcija, u C++ je uveden koncept inline funkcije **(funkcije ugrađene direktno u kod programa)**:

  ```
  inline int zbir(int a,int b) {return a+b;}
  ```

- Prethodni primjer predstavlja zaista jednu kratku funkciju čiji bi poziv **(i povratak iz funkcije)** odnio više vremena nego izvršavanje. Stoga je dodata ključna riječ **inline** koja znači da se umjesto poziva funkcije vrši postavljanje koda funkcija na svakom mjestu gdje se funkcija poziva. Ovo dovodi do bržeg izvršavanja, ali nešto veće EXE verzije programa. Da se EXE verzija ne bi nepotrebno uvećavala, kao inline se deklarišu samo relativno kratke funkcije.

Poboljšanja kod funkcija

- Ako kompajler iz nekog razloga ne može da kreira inline funkciju, odnosno da je ugrađuje u kod, kreiraće je kao standardnu funkciju. Ostala pravila kod inline funkcija su ista kao i kod standardnih funkcija.
- Drugi važan koncept uveden kod funkcija u C++, a koji ne postoji u C-u, su **podrazumijevani argumenti**.
- Pretpostavimo jednostavan program koji treba da računa težinu tijela na osnovu njegove mase:
  float tezina(float masa) {return 9.81*masa;}
- To će dobro raditi u našoj zemlji, ali što ako nam se desi da ponekad proračun vršimo za neku drugu lokaciju (recimo, Meksiko Siti gdje je gravitaciono ubrzanje manje ili za neki treći grad)?

Podrazumijevani argumenti f-ja

- Prethodni problem se može riješiti pisanjem mnoštva funkcija, gdje bi svaka radila sa svojim g ili pisanjem jedne funkcije koja bi imala dva argumenta - masa i g. Drugi način je bolji, ali ne i idealan, jer bi stalno funkciju pozivali kao tezina(m,9.81), odnosno u većini slučajeva se nepotrebno zamarali sa pisanjem 9.81, koje je uobičajeno.
- C++ dozvoljava sljedeći oblik f-je:
  float tezina(float masa,float g=9.81) {return g*masa;}

Podrazumijevani (default) argument.

Poziv f-je: tezina(m) vraća 9.81*m, a ako se pozove tezina(m,g) vraća g*m.

Default argumenti - Pravila

- Dakle, ako se izostavi podrazumijevani argument, onda se uzima podrazumijevana vrijednost iz zaglavlja f-je.
- Funkcija može imati više podrazumijevanih argumenata, ali svi **moraju biti na kraju**. Npr.
  int fun(int a,int b=2,int c=3) {return a+b*c;}
  Poziv fun(x) se tretira kao fun(x,2,3), a poziv fun(x,y) se tretira kao fun(x,y,3).
- Najveće unaprijeđenje u pogledu f-ja C++ je ostvario **preklapanjem** (eng. overloading) imena **funkcija**.

Preklapanje funkcija

- Mnoge funkcije rade prirodno slične stvari. Recimo, maksimum dva cijela broja je slična funkcija kao maksimum dva realna broja, a ova je slična kao maksimum niza realnih brojeva.
- U programskom jeziku C smo realizovali funkcije po pravilu jedan problem - jedna funkcija, odnosno (što je gora formulacija) više sličnih problema - više sličnih funkcija.
- C++ nam dozvoljava da više funkcija koje rade slične stvari nazovemo istim imenom. Taj koncept se naziva preklapanje **imena** funkcija – function overloading.
- Koja od preklopljenih funkcija će se izvršiti zavisi od argumenata sa kojima je pozvana.

Preklapanje imena f-ja - Primjer

- Kreirajmo tri preklopljene f-je:

```
int max(int a,int b) {return (a>b) ? a : b;}
float max(float a,float b){return (a>b) ? a : b;}
float max(float *a,int n) {
        int max=a[0];
        for(int i=1; i < n; i++)
                if(a[i]>max)
                        max=a[i];
        return max;}
```

→ **Tip podatka float se mnogo manje koristi u C++ u odnosu na double.**

- Poziv max(3,5) poziva prvu funkciju jer postoji potpuno slaganje argumenta, max(2.4,3.1) poziva drugu, dok kod: float x[50]; int n; /*...neke naredbe...*/ max(x,n); poziva treću.

Razrješenje poziva

- Prethodni primjer je demonstrirao situaciju kada postoji potpuno poklapanje argumenata pozvane funkcije sa pojedinim njenim realizacijama.
- Postoji problem što se dešava ako takvog poklapanja nema.
- Procedura koja se provodi i koja odlučuje koja se funkcija zapravo poziva naziva se razrješenje poziva.
- Grubo pravilo: za svaku funkciju datog imena broji se koliko ima potpunih preklapanja sa tipovima zadatih argumenata. Ako postoji jedna funkcija koja ima najviše preklapanja poziva se ona, a ako ima više od jedne sa najvećim brojem preklapanja provjerava se koja od njih ima najveći broj preostalih argumenata koji se imlicitnim konverzijama mogu prevesti u tražene argumente. Ako se nađe jedna funkcija - poziva se ona, a ako ima više od jedne sa podjednako kvalitetnim argumentima dolazi do prekida izvršavanja programa i greške usljed dvosmislenosti.

Razrješenje poziva

- Npr. poziv max(2,2.5) će dovesti do greške pošto dvije funkcije imaju po jedno potpuno poklapanje argumenta i jedno koje se standardnim implicitnim konverzijama može izvršiti.
- Razrješenje poziva se dodatno komplikuje ako imamo funkcije sa podrazumijevanim argumentima.
- Postupak razrješenja poziva je detaljno analiziran u knjizi od Milićeva na stranama 199-203.
- Mi ćemo to ovdje preskočiti.
- Problem postaje dodatno usložnjen ako kompajler ne prati preporuke ANSI komiteta. Npr. naš kompajler će prilikom poziva max(2,2.5) pozvati funkciju koja prima dva cijelobrojna podatka dok će Borland pozvati funkciju sa dva float-a, jer to doživljava kao bolju situaciju u kojoj ne dolazi do odsjecanja necjelobrojnog dijela prilikom prenosa argumenta.

Razrješenje poziva - Savjet

- Najbolje je da do pomenutih problema u razrješenju poziva i ne dođe.
- Naime, ako veoma slične funkcije realizujemo kao što smo to uradili sa prve dvije, mi obije operacije možemo odraditi sa drugom funkcijom, koja sa standardnom konverzijom može da primi i argumente koji su cijeli brojevi i da, ako su argumenti cijeli brojevi, vrati rezultat koji je suštinski cijeli broj. Stoga bi u našem sistemu izbrisali prvu funkciju i ne bi imali problem sa razrješenjem.
- Brižljivim dizajnom funkcija izbjećićete problem tumačenja razrješenja poziva funkcija.

Razrješenje poziva

- Ako funkcije imaju iste argumente, a različit tip rezultata, doći će uvijek do greške u kompajliranju **(ili greške u izvršavanju)** pošto se te dvije funkcije ne mogu razdvojiti argumentima i nikada se ne može protumačiti koja će se pozvati!!!!
- Funkcije sa istim imenima moraju se razlikovati barem po jednom argumentu!!!
- U C-u smo vidjeli da ime funkcije može poslužiti kao njena adresa. To nije bio problem jer je funkcija morala da ima jedinstveno ime i to ime je moglo da se upotrijebi kao pristupna adresa kodu funkcije.
- U C++ više funkcija može da dijeli isto ime, pa se, recimo, njena adresa mora specificirati i tipom argumenata. Ako je adresa funkcije argument neke druge funkcije, onda se mora slagati po tipu pokazivača **(pokazivač na cjelobrojnu funkciju se ne može implicitno konvertovati u pokazivač na neku drugu funkciju)**, a mora postojati i jedinstvena identifikacija argumenata funkcije. Npr. zaglavlje funkcije koja ima za argument drugu funkciju bi moglo biti: int fja(int (*max)(int,int)).

Komentari i konstante

- Pored komentara u C-u koji počinje sa /* i završava sa */ i može se prostirati preko više redova, C++ ima i komentar oblika // koji važi do kraja reda (nije postojao u C-u prije C99 kada je usvojen iz C++).
- Kod konstanti nema razlika u sintaksi: const int i=1; predstavlja konstantu čiji pokušaj promjene dovodi do greške u izvršavanju ili kompajliranju. Pored toga, uveli smo već konstantne pokazivače:

```
const int *p;
int * const d;
const int * const t;
```

U prvoj varijanti p je pokazivač na konstantni cijeli broj. U drugoj varijanti d je konstantni pokazivač na cijeli broj (koji se može mijenjati). U trećoj varijanti i t (pokazivač) i broj na koji t pokazuje su konstantni.
Memo tehnika: ukinuti const i protumačiti deklaraciju, vratiti const i pogledati što se nalazi desno od const-a (to je zapravo konstantno).

Konstante - Pretprocesor

- Pa gdje se konstante razlikuju? U upotrebi. U C++ se mnogo više koriste da bi se umanjila upotreba pretprocesora. Ako je argument funkcije const **(obično se primjenjuje kod pokazivača)** onda ono na što taj argument pokazuje **(ili referencira o čemu ćemo kasnije)** ne može biti promijenjeno u funkciji, tj. u slučaju promjene dolazi do prekida izvršenja programa
- Kakve su razlike kod pretprocesora kod C-a i C++? Nikakve **(ili gotovo nikakve).** Razlika je samo u upotrebi.
- Osnovna razlika je da se znatno manje koristi makro naredba #define.

Upotreba #define

- Pretprocesorska direktiva #define u C-u je imala tri namjene:
  - markozamjena (npr. #define MAX 100, kojom se svaka pojava MAX u kodu programa prije početka kompajliranja mijenja sa 100. Problem je bila promjena na slijepo, jer se MAXMAX mijenjalo sa 100100, ali i to što se na ovaj način u izrazima gdje se makro MAX pojavljuje ne može vršiti provjera ispravnosti tipova. Umjesto ovoga koristiti globalnu konstantu: const int MAX=100;).
  - makrorazvoj (npr. #define zbir(a,b) a+b koji je svako pojavljivanje makro zbir mijenjao sa odgovarajućim zbirom. Ovo je bio model jednostavne funkcije koja se kompajlirala bez provjere tipova i nosila je niz sitnih problema tipa zbir(x,y)/2 koje je mijenjano sa x+y/2. Umjesto ovoga koristiti inline funkcije koje pored ostalog daju i kontrolu upotrebe tipova podataka:
    inline int zbir(int a,int b) {return a+b;}

Upotreba #define

- Treća upotreba #define je preživjela:
  - makrodefinicija tipa #define _MAX. Na ovaj način se u program uvodi ime _MAX. Nije pravilo već praksa da imena makroa počinju sa podvlakom. Obično se navodi u biblioteci klasa ili funkcija. Npr. napravimo biblioteku klasa ili funkcija koja koristi prvu biblioteku. Zatim pišemo program koji uključuje obije biblioteke preko naredbe #include. Programer koji koristi biblioteke ne mora da zna kako su one kreirane. Međutim, pošto su funkcije ili klase prve biblioteke uključene i u drugu biblioteku došlo bi do prekida programa jer imamo dva puta definisane funkcije sa istim imenom i istim argumentima (ili dvije klase sa istim nazivom), što nije dozvoljeno.

Korišćenje definisanih makroa

- Programi često imaju pitanja tipa:

```
#ifndef _MAX
#include <bibl_sa_max.h>
#endif
```

Ako makro _MAX nije definisan do sada uključi biblioteku. Selekcije u ovom dijelu uvijek završavaju sa #endif.

Podsjetite se i drugih makroselekcija tipa #ifdef (ako je definisan makro), #undef kojom neki makro prestaje da bude definisan itd.

- #define ima još neke primjene koje mogu da se koriste i u savremenim C kompajlerima, a standardizovane su tek nakon pojave C++.
- To su tzv. specijalni tipovi makrorazvoja.

Specijalni tipovi makrorazvoja

- Prvi specijalni tip makrorazvoja je:
  #define napisistring(x) puts(#x)
  U programu: napisistring(ssss) se mijenja sa naredbom puts("ssss").

Taraba zapravo znači da se argument makrorazvoja tretira kao string.

- Drugi specijalni tip makrorazvoja je:
  #define abc(x) R##x##C
- Kao konačni dodatak priči o pretprocesoru kažimo da savremeni kompajleri dozvoljavaju i korišćenje određenih ugrađenih makrokonstanti: _ _DATE_ _ (tekući datum; da, u pravu ste, makrou prethodi i za njim slijede dvije podvlake), _ _TIME_ _ (tekuće vrijeme), _ _LINE_ _ (linija koda), _ _FILE_ _ (fajl koji se izvršava).

Argument makroa može služiti za formiranje različitih identifikatora, recimo R1C, R2C itd.

Stringovi

- Programski jezik C++ dozvoljava neke neobične operacije kod stringova. Npr. dozvoljeno je napisati “abc”+”123” što će rezultirati konkatenacijom stringova “abc123”.
- Pored toga, string se tretira kao pravi podatak, a ne samo kao pokazivač na niz karaktera. Ovaj detalj ipak ima dosta veze sa OO orjentacijom C++, ali o tome kasnije.
- Mnogi kompajleri potiskuju korišćenje niza karaktera kao tzv. stringa sa terminacionim simbolom u korist klasa koje realizuju stringove, ali to ćemo moći da obradimo kasnije u našem kursu.

Formatizovani I/O

- C i C++ imaju osobinu da ulaz/izlaz nijesu sastavni dio definicije programskog jezika.
- U C-u smo intenzivno koristili stdio.h biblioteku sa naredbama printf i scanf.
- Ako u C++ radimo sa konzolskim aplikacijama **(vjerujte mi da je ozbiljno programiranje 90% rad na konzoli)** i dalje možemo da koristimo ovu biblioteku, ali se zbog OO orjentacije preporučuje korišćenje iostream.h.
- Napominjemo da po standardu C++ nije potrebno naglašavati .h za najčešće korišćene biblioteke u pretprocesorskoj direktivi #include, ali da veći broj kompajlera .h podržava, a neki na tome još uvijek insistiraju.

iostream.h

- Naredbe **(nijesu naredbe, ali ih za početak možemo tako zvati)** za formatizovani ulaz/izlaz u biblioteci iostream.h su: cin (čita se si-in) i cout (čita se si-aut).
- cin se koristi za standardni ulaz:
  cin>>a>>b>>c;
- Na ovaj način se ono što se unese sa tastature smiješta redom u promjenljive a, b, c koje mogu biti bilo kog tipa podataka uključujući i string (pokazivač na niz karaktera).
- Znači, ne koristimo adresu kao kod scanf-a i ne objašnjavamo kojeg je tipa podatak ("%d").

Naredba cout

- Naredba cout se koristi za standardni izlaz:
  cout<<a<<"Neki tekst"<<2<<a+2<<'\n'<<endl;
- Na ovaj način smo odštampali promjenljivu a, zatim string "Neki tekst", broj 2, vrijednost izraza a+2, prešli u novi red i konačno sa endl ponovo prešli u novi red (endl je skraćenica od kraj linije - end of line).
- Za nekoliko nedjelja shvatićemo da cin i cout nijesu naredbe i da >> i << imaju sasvim specifično značenje.

Alokacija i dealokacija

- Niko nam ne brani da i dalje koristimo naredbe malloc i free iz standardnog zaglavlja alloc.h.
- Međutim, programski jezik C++ posjeduje naredbe (u stvari operatore ugrađene u jezik za čije korišćenje nije potrebna programska biblioteka) koje bolje podržavaju OO metodologiju: new (za alokaciju) i delete (za dealokaciju).
- int *p = new int; //zauzima memoriju za jedan cijeli
  //broj na koji pokazuje *p*
- int *p=new int(3); //isto kao gore, uz inicijalizaciju broja na
  //koji pokazuje *p* na cijeli broj 3
- int *p=new int[50]; //alocira memoriju za 50 cijelih brojeva,
  //a *p* pokazuje na početak niza

Alokacija i dealokacija

- Alokacija je, školski rečeno, obavezno praćena provjerom alokacije, ali, grubo govoreći, savremeni C++ kompajleri dozvoljavaju adresiranje ogromnog adresnog prostora i ako alokacija ne uspije, nema šanse ni da vam ostatak koda dobro radi, jer su vjerovatno gotovo svi sistemski resursi potrošeni. Ipak, uvijek vršite provjeru ispravnosti alokacije.
- Dealokacija se vrši sa:
  delete p; //ako je u pitanju podatak na koji pokazuje p
  delete [] p; //ako je u pitanju niz
- Često se *p* postavlja na NULL (ili 0) nakon dealokacije da bi sugerisao ostatku koda da u tom trenutku ne pokazuje na podatak i da ne bi pamtila adresu memorije koja je upravo oslobođena.

Reference

- Naučuli smo: funkcija se može pozvati po vrijednosti (call by value) i po referenci.
- Naučili smo: programski jezik C ne posjeduje referencu, pa se poziv po referenci simulira preko pokazivača.
- Stariji programski jezici (recimo Pascal) posjeduju referencu.
- Programski jezik C++ pored pokazivača posjeduje referencu.
- Pa što je onda referenca?

Referenca u C++

- Referenca je drugo ime za neki memorijski objekat (ponekad se naziva alijas, npr. Muhamed Ali alijas Kasijus Klej).
- Referenca se može koristiti i u dijelu glavnog programa:
  int &j = i;
  j = 3;
- Nakon ovog koda j=3, ali i i=3, jer su to samo dva imena za isti objekat.
- Uočimo da referenca mora biti inicijalizovana!!! nekim memorijskim objektom, za razliku od pokazivača koji ne mora.
- Dalje, pristup referenci ne zahtjeva navođenje operatora (nijesmo pisali &j=3).

Reference i pokazivači

- Referenca ne može da promijeni objekat na koji se odnosi, niti da pokaže na NULL objekat. Npr.

```
int i=3;
int &j = i, k = 2;    //j=3 jer je isto kao i
j = k;                //i=2 jer je isto što i j
i=4;                  //j=4, ali k i dalje ostaje 2, jer nije to j
```

- Referenca se rijetko koristi u programu **(jedino ako jedan dio programera koji rade na programu neku promjenljivu zovu na jedan način, a drugi dio na drugi način)** već je prevashodna primjena kod funkcija.

Reference i funkcije

- Npr. dio glavnog programa: int a = 2, b = 3; fun(a,b);
- Neka je funkcija: void fun(int i,int &j) {i++; j++}
- Očigledno je da se pokazivač i referenca mogu koristiti za slične namjene.
- Postoje sljedeća pravila:
  - Ako promjenljiva može da pokaže na NULL ili može da promjeni promjenljivu na koju ukazuje koriste se pokazivači.
  - Ako je objekat memorijski zahtjevan treba koristiti referencu jer ne dolazi do kopiranja objekta za potrebe funkcije, već funkcija samo koristi drugo ime za isti objekat da bi mu pristupila. Štedi se i u dealokaciji ovakvih objekata.

j je u funkciji samo drugo ime za promjenljivu *b* iz glavnog programa, stoga promjena *j* u funkciji ima uticaj na promjenu b u glavnom programu. b=4 nakon izvršavanje fun

Referenca kao rezultat funkcije

- Funkcija može da vrati rezultat koji je referenca. Npr.
  int &f(int &i) {return i;}

Ova oznaka znači da funkcija vraća kao rezultat memorijski objekat koji mora biti alociran u pozivajućem modulu (recimo u glavnom programu).

- Primjer glavnog programa:
  void main(){int x; f(x)=1;}
- Ovo je nevjerovatan poziv. Suprotno onome što smo do sada učili, funkcija (njen rezultat) može biti na lijevoj strani ako funkcija vraća referencu. U ovom slučaju funkcija vraća argument funkcije (x) i to stvarni objekat tako da je rezultat ove operacije x=1;. Za sada je ovo samo atrakcija, ali ćemo ovo upotrebljavati dosta kasnije u našem kursu. Već za nekoliko nedjelja koristićemo rezultat koji je referenca da bi uštedjeli na kopiranju objekata prilikom vraćanja rezultata.

Reference – Dodatna pravila

- Ne može se deklarisati:
  - referenca na referencu;
  - referenca na polje bitova;
  - referenca na void;
  - pokazivač na referencu.

switch-case - enum

- U C-u, argument naredbe switch mogao je biti samo cijeli broj (ili cjelobrojni izraz ili nešto što se može konverzijama dovesti do cijelog broja). U C++ argument naredbe switch može biti bilo koji tip podatka.
- Nabrajanje (enumeracija) ima istu strukturu i ista pravila kao u C-u. Npr.
  enum color {bijelo=0, crno=7, zeleno=3, plavo=1};
- Osnovna razlika je da ime nabrajanja koje po pravilu nijesmo ni navodili ni koristili u C-u (u ovom slučaju color) ovdje po pravilu navodimo i često koristimo. Naime, color je sada pored skupine cjelobrojnih konstanti i pravi tip podatka (moguće je deklarisati color a;) koji može uzeti samo navedene vrijednosti i koji može biti argument i rezultat funkcije: color fun(color x,int a){/**/}.

exit i abort

- Funkcija exit (standardno zaglavlje stdlib.h) zatvara sve otvorene fajlove i vraća cjelobrojnu konstantu kao rezultat operativnom sistemu. U C++ ova funkcija pored prekidanja programa i navedenih operacija poziva funkciju atexit na koju može da utiče programer.
- Alternativno može se koristiti funkcija abort koja prekida izvršavanje programa bez mogućnosti intervencije i bez obavještavanja OS-a.
- Ovim smo završili pregled razlika prevashodno neobjektne prirode između C-a i C++-a. Od naredne nedjelje uplovljavamo u vode OOP.