Objektno orijentisano programiranje 1

Proširenja jezika C

Deklaracije i definicije

- *Deklaracija* je iskaz koji samo
  - uvodi neko ime (identifikator) u program
  - govori prevodiocu kojoj jezičkoj kategoriji pripada ime
- Ime se može koristiti samo ako je prethodno barem deklarisano
- *Definicija* je ona deklaracija koja
  - kreira objekat (dodeljuje mu memorijski prostor),
  - navodi telo funkcije, ili
  - u potpunosti navodi strukturu korisničkog tipa (uključujući klase u C++)
- U programu može postojati:
  - samo jedna definicija jednog objekta, funkcije i tipa
  - proizvoljno mnogo deklaracija
- Na jeziku C definicije objekata mogu biti samo na početku bloka, a njihova oblast važenja je do kraja bloka
- Na jeziku C++ definicija objekta je naredba, pa se može naći bilo gde u programu
- Objekat može biti inicijalizovan u definiciji

Objekti

- Objekat u širem smislu (podatak):
  - definisano područje u memoriji, u toku izvršavanja programa
  - primerak proizvoljnog tipa (ugrađenog ili korisničkog)
- Objekat u užem smislu:
  - primerak (instanca) klase
- Objekat je nešto (u memoriji) što ima stanje, ponašanje i identitet
  - funkcija je u memoriji, ali nije objekat (nema stanje)
  - podatak primitivnog tipa (koji je objekat u širem smislu) nema identitet
- Promenljiva:
  - lokacija u kojoj se čuva podatak koji nije konstantan
- Promenljiva može biti:
  - globalna ili lokalna,
  - statička, automatska, dinamička, privremena (tranzijentna)

Lvrednosti

- *lvrednost* (*lvalue*) je izraz koji upućuje na objekat (u širem smislu) ili funkciju
- *lvalue* je kovanica od *left-value*
  "nešto što može da stoji sa leve strane znaka dodele vrednosti"
- Sa leve strane znaka = mogu da stoje samo promenljive *lvrednosti*
- Promenljiva *lvrednost* (*modifiable lvalue*) je ona *lvrednost*,
  koja nije ime funkcije, ime niza, ni konstantni objekat
- Za svaki operator se definiše:
  - da li zahteva kao operand(e) *lvrednost*(i) i
  - da li vraća *lvrednost* kao rezultat
- Operatori čiji operandi moraju da budu *lvrednosti*:
  - unarni `&`, `++` i `--`, kao i levi operandi svih operatora dodele
- Operatori čiji su rezultati *lvrednosti*:
  - unarni `*`,`[]`, prefiksni `++` i `--`, kao i operatori dodele
- *Dvrednost* (*rvalue – right value*) je izraz koji nije *lvrednost*.

Primeri lvrednosti

```
int i=0;          // i je lvrednost
int *p=&i;        // p je lvrednost
*p=7;             // *p je lvrednost

int *q[100];
q[10]=&i;         // q[10] je lvrednost
*q[10]=1;         // *q[10] je lvrednost
q=&i;             // ! GRESKA: ime niza nije promenljiva lvrednost

int a=1,b=2,c=3;
(a=b)=c;          // (a=b) je lvrednost
(a+b)=c;          // ! GRESKA:(a+b) nije lvrednost
++ ++i;           // ++(++i)
i++ ++;           // ! GRESKA: postinkrement ne daje lvrednost

long x=5; int y=0,z=0;
(x?y:z)=1;        // x?y=1:z=1;
(x?10:z)=2;       // !GRESKA: 10 nije lvrednost
```

Oblast važenja (doseg)

- Oblast važenja (doseg, *scope*) imena:
  - onaj deo teksta programa u kome se deklarisano ime može koristiti
- Globalna imena:
  - imena koja se deklarišu van svih funkcija i klasa
  - oblast važenja: deo teksta od mesta deklaracije do kraja datoteke
- Lokalna imena:
  - imena deklarisana unutar bloka, uključujući i blok tela funkcije
  - oblast važenja: od mesta deklarisanja, do završetka dotičnog bloka
- Sakrivanje imena:
  - ako se redefiniše u unutrašnjem bloku,
    ime iz spoljašnjeg bloka je sakriveno do izlaska iz unutrašnjeg
- Pristup sakrivenom globalnom imenu:
  - navođenjem operatora `::` ispred imena
- Pristup sakrivenom imenu spoljašnjeg bloka nije moguć

Primer dosega i sakrivanja

```
int x=0;              // globalno x
void f () {
  int y=x,            // lokalno y, globalno x;
      x=y;            // lokalno x, sakriva globalno x
  x=1;                // pristup lokalnom x
  ::x=5;              // pristup globalnom x
  {
    int x;            // lokalno x, sakriva prethodno x
    x=2;              // pristup drugom lokalnom x
  }
  x=3;                // pristup prvom lokalnom x
}
int *p=&x;            // uzimanje adrese globalnog x
```

Neki specifični dosezi

- U naredbi `for` umesto `izraz0` u jeziku C:
  ```
  for(izraz0; izraz1; izraz2)
  ```
  u jeziku C++ nalazi se naredba
  - može da bude definicija promenljive
- Po standardu, promenljiva definisana na taj način (brojač petlje) je lokalna promenljiva `for` naredbe
- Neki prevodioci (npr. MS VC++ v.6) takvu promenljivu smatraju definisanom za blok u kome se nalazi `for`
- U uslovu `if` se može definisati celobrojna ili pokazivačka promenljiva
  - doseg je do kraja then, odnosno `else` bloka
    ```
    if (int k=i+j){…}else{…}
    ```
- Formalni argumenti funkcije:
  - kao lokalne promenljive deklarisane u bloku tela funkcije:
    ```
    void f (int x){int x;} // ! GRESKA
    ```

Primer dosega brojača petlje `for`

```
for (int i=0; i<10; i++) {
    if (a[i]==x) break;
    //...
}

if (i==10)    //! GRESKA (po standardu)
    // u MS VC++ v.6 moze se pristupati imenu i

for (int i=9; i>=0; i--) cout<<a[i];
    // u MS VC++ v.6 ovo je greska, i je vec definisano
```

Doseg strukture/klase i funkcije

- Oblast važenja strukture/klase imaju svi njeni članovi
  - to su imena deklarisana unutar definicije strukture/klase
- Imenu koje ima oblast važenja klase, van te oblasti, može se pristupiti preko operatora:
  - `.`, gde je levi operand objekat,
  - `->`, gde je levi operand pokazivač na objekat,
  - `::`, gde je levi operand ime klase
- Oblast važenja funkcije imaju samo labele (za `goto` naredbe)
  - labele se mogu navesti bilo gde unutar tela funkcije, a u dosegu su u celoj funkciji

Primer dosega klase

```
class X {
   public:              // ili:
     int x;             // void f();
     void f();          // int x;
};
void X::f () {...x...} // :: proširenje dosega
void g(){
  X xx, *px;
  px=&xx;
  xx.x=0;     // moze: xx.X::x;   ali nema potrebe
  xx.f();     // moze: xx.X::f(); ali nema potrebe
  px->x=1;
  px->f();
}
```

Životni vek objekata

- Životni vek objekta: vreme u toku izvršavanja programa u kojem objekat postoji i za koje mu se može pristupati
- Na početku životnog veka, objekat se kreira
  - poziva se njegov konstruktor, ako ga ima
- Na kraju životnog veka se objekat uništava
  - poziva se njegov destruktor, ako ga ima

Vrste objekata po životnom veku

- Po životnom veku, objekti se dele na:
  - statičke
  - automatske
  - dinamičke
  - tranzijentne (privremene)
- Vek atributa klase = vek objekta kojem pripadaju
- Vek parametra = vek automatskog objekta
  - inicijalizuju se vrednostima stvarnih argumenata
  - semantika ista kao kod inicijalizacije objekta u definiciji

Statički i automatski objekti

- *Automatski* objekat je lokalni objekat koji nije deklarisan kao `static`
  - životni vek: od njegove definicije, do napuštanja oblasti važenja
  - kreira se iznova pri svakom pozivu bloka u kome je deklarisan
  - prostor za automatske objekte se alocira na *stack*-u
- *Statički* objekat je globalni objekat ili lokalni deklarisan kao `static`
  - životni vek: od izvršavanja definicije do kraja izvršavanja funkcije `main()`
  - globalni statički objekti
    - kreiraju se samo jednom, na početku izvršavanja programa
    - kreiraju se pre korišćenja bilo koje funkcije ili objekta iz istog fajla
    - nije obavezno da se kreiraju pre poziva funkcije `main()`
  - lokalni statički objekti
    - počinju da žive pri prvom nailasku toka programa na njihovu definiciju

Primer

```
int a=1;
void f() {
  int b=1;    // inicijalizuje se pri svakom pozivu
  static int c=1;  // inicijalizuje se samo jednom
  cout<<"a="<<a++<<" b="<<b++<<" c="<<c++<<endl;
}

void main() {
  while (a<3) f();
}
```

izlaz:

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
```

Dinamički i privremeni objekti

- *Dinamički* objekti se kreiraju i uništavaju posebnim operacijama
  - životni vek dinamičkih objekata neposredno kontroliše programer
  - oni se kreiraju operatorom `new`, a ukidaju operatorom `delete`
  - prostor za dinamičke objekte se alocira na *heap*-u

- *Privremeni* objekti se kreiraju pri izračunavanju izraza
  - životni vek privremenih objekata je kratak i nedefinisan
  - privremeni objekti služe za:
    - odlaganje međurezultata
    - privremeno smeštanje vraćene vrednosti funkcije
  - najčešće se uništavaju čim više nisu potrebni

Leksički elementi

- Komentari:
  - `/* ... */` - u više redova, uveo jezik C
  - `// ...` - do kraja reda, uveo jezik C++
- Ključne reči:
  - 73 (29 više nego u C) + 11 za alternative operatora (npr. `and`, ...)
  - neke ključne reči u C i C++ su različite, sa istim značenjem C++11: `bool`, C11: `_Bool`
  - specijalne (nisu rezervisane) reči (specifikatori): `final` i `override`
- Identifikatori:
  - konvencija: ne treba započinjati sa _ i __
    - imena u biblioteci C počinju _
    - imena u biblioteci C++ počinju __

Tipizacija

- Stroga tipizacija:
  - svaki objekat ima svoj tačno određeni tip
  - objekti različitih tipova se ne mogu proizvoljno zamenjivati
- C++ je hibridan jezik:
  - u manipulisanju primitivnim tipovima je labavo tipiziran
  - u manipulisanju klasnim tipovima je strogo tipizirani jezik, što je u duhu njegove objektne orijentacije
- Konverzija tipa:
  - ako se na nekom mestu očekuje objekat jednog, a koristi se objekat drugog tipa, potrebna je *konverzija*

```
char f(float i, float j) { /* ... */ }
int k=f(5.5,5); // najpre konverzija (float)5,
                // a posle konv. rez. char u int
```

Kada je potrebna konverzija?

- Slučajevi kada je potrebno vršiti konverziju su:
  - operatori za ugrađene tipove
    zahtevaju operande odgovarajućeg tipa
  - naredbe (`if`, `for`, `do`, `while`, `switch`)
    zahtevaju izraze odgovarajućeg tipa
  - pri inicijalizaciji objekta jednog tipa pomoću objekta drugog tipa
    - pri definisanju objekata i njihovoj inicijalizaciji
    - pri pozivu funkcije,
      kada su stvarni argumenti drugačijeg tipa od formalnih argumenata
    - pri povratku iz funkcije,
      ako je izraz iza return drugačijeg tipa od tipa rezultata funkcije
      - privremeni objekat koji prihvata vrednost funkcije
        se inicijalizuje vrednošću izraza iza return

Vrste konverzije tipova

- Konverzija tipa može biti:
  - *standardna* – ugrađena u jezik ili
  - *korisnička* – definiše je programer za svoje tipove
- Standardne konverzije su, na primer:
  - konverzije iz tipa `int` u tip `float`, ili iz tipa `char` u tip `int`, i sl.
- Konverzija tipa može biti:
  - *implicitna* – prevodilac je automatski vrši, ako je dozvoljena
  - *eksplicitna* – zahteva programer
- Jedan način zahtevanja eksplicitne konverzije:
  - pomoću C operatora kast (*cast*): `(`*`tip`*`)`*`izraz`*
- Jezik C++ uvodi 4 specifična kast operatora
- Postoji i drugi mehanizam konverzije (konverzioni konstruktor)

Pridruživanje imena tipu

- U jeziku C (i nasleđeno u C++):

```
typedef opis_tipa ime_tipa
```

- U jeziku C++ i drugi (čitljiviji) način:

```
using ime_tipa = opis tipa;
```

- Primer:

```
typedef unsigned long long int Ceo1;
using Ceo2 = unsigned long long int;

typedef int Niz1[10];
using Niz2 = int [10];

typedef int (*PFun1) (int,int);
using PFun2 = int (*) (int,int);
```

Određivanje tipa izrazom

- Tip se može odrediti i naredbom:
  `decltype (izraz1) promenljiva [= izraz2];`
- Tip promenljive će biti jednak tipu `izraz1`
- Tip `izraz1` se ne izračunava, a tip `izraz2` nije bitan
- Po potrebi se tip izraza `izraz2` konvertuje u tip `izraz1`
- Primer:

```
int x=1;
double y=1.5;
decltype(x) a=x;          // int a
decltype(++x+y) b=++x+y;// double b   b==3.5
decltype(y) c=2;          // double c   c==2.0
decltype(x) d=2.5;        // int d      d==2
```

- Primena – kod šablona

Automatsko određivanje tipa

- Modifikator `auto` u jeziku C – automatska promenljiva (neobavezno)
- U jeziku C++ nije dozvoljeno takvo korišćenje

```
auto int a = 10;  // ! GRESKA
```

- U C++: automatsko određivanje tipa na osnovu izraza inicijalizatora
- Sintaksa: **`auto`** `promenljiva = izraz`
- Primer:

```
int x=1;
double y=2.5;
auto a = x;       // int a
auto b = ++x + y; // double b
auto c = &a;      // int *c
```

Odloženo navođenje tipa funkcije

- Umesto uobičajenog povratnog tipa – ključna reč `auto`
- Tip se navodi iza parametara, u deklaraciji ili definiciji

  `auto ime(parametri) -> tip telo`
- Koristi se kod šablonskih funkcija
- C++14: deo `-> tip` može da se izostavi
  - u tom slučaju će se tip odrediti na osnovu tipa izraza u naredbi `return`
  - ako se `-> tip` izostavi u deklaraciji, tip se određuje tek na osnovu definicije
  - f-ja ne sme da se poziva pre navođenja definicije na mestima gde je tip bitan
- Primer:

```
auto f(){return 1;}      // tip rezultata je int
auto g();                // tip rezultata neodredjen
auto a=g();              // ! GRESKA
auto g(){return 0.5;}    // tip rezultata je double
auto b=g();              // tip b je double
```

Konstante

- Konstantni tip je izvedeni tip
  - dobija se iz nekog osnovnog tipa pomoću specifikatora `const`
  - zadržava sve osobine osnovnog tipa, samo se podatak ne može menjati
  - primeri: `const float pi=3.14f; const char plus='+';`
- Konstanta mora da se inicijalizuje pri definisanju
  - može i izrazom koji nije konstantan: `int i=10; const int c=i;`
- Prevodilac često ne odvaja memorijski prostor za konstantu
- Konstante inicijalizovane konstantnim izrazom mogu da se koriste u izrazima koji moraju biti konstantni (koje prevodilac mora da izračuna u toku prevođenja)
  - na primer, konstante mogu da se koriste u izrazima koji definišu dimenzije nizova
- Umesto simboličkih konstanti koje se uvode sa `#define` preporuka je koristiti tipizirane konstante koje se uvode sa `const`
- Dosledno korišćenje konstanti u programu obezbeđuje podršku prevodioca u sprečavanju grešaka

Konstante i pokazivači

- Pokazivač na konstantu: reč `const` ispred cele definicije
- Konstantni pokazivač: reč `const` ispred imena pokazivača

```
char niz[]={'a','s','d','f','g','h','\0'};
const char *pk=niz;      // pokazivac na konstantu
pk[3]='a';               // ! GRESKA
pk="qwerty";             // ispravno

char *const kp=niz;      // konstantni pokazivac
kp[3]='a';               // ispravno
kp="qwerty";             // ! GRESKA

const char *const kpk=niz;
kpk[3]='a';              // ! GRESKA
kpk="qwerty";            // ! GRESKA
```

Konstante i funkcije

- `const` ispred formalnog argumenta koji je pokazivač, obezbeđuje da fukcija ne menja objekat:

```
char *strcpy(char *p, const char *q);
```

- `const` ispred tipa rezultata funkcije, obezbeđuje nepromenljivost privremenog objekta rezultata
- Za vraćenu vrednost koja je pokazivač na konstantu, ne može se preko vraćenog pokazivača menjati objekat:

```
const char* f();
*f()='a';          // ! GRESKA
```

Konstantni podaci

- Konstantan izraz - vrednost može da se izračuna za vreme prevođenja
- Svi operandi u konstantnim izrazima moraju biti konstantni
- Vrednost ovakvog izraza može da se koristi tamo gde se zahteva konstanta
- Konstantan podatak se uvodi modifikatorom `constexpr`
- Inicijalizator konstantnog podatka mora biti konstantan izraz
- Konst. podatak je statički i ima unutrašnje povezivanje (implicitno `static`)
- Razlika u odnosu na nepromenljivi podatak koji se uvodi sa `const`
- Primer:

```
int a=1;              // promenljiv podatak (promenljiva)
const int b=a;        // nepromenljiv podatak
const int C=1;        // imenovana (simbolicka) konstanta
constexpr int d=1;  // konstantan podatak
constexpr int e=d+C;
constexpr int f=a;  // ! GRESKA
constexpr int g=b;  // ! GRESKA
```

Konstantne funkcije

- Deklariše se i definiše modifikatorom `constexpr` na početku
- Sugestija prevodiocu da izračuna rezultat funkcije u toku prevođenja
- Rezultat je konstantan podatak
- Ako na mestu poziva ne može da se izračuna vrednost – nije greška
  - tada rezultat nije konstantan podatak
- Ne smeju da imaju bočne efekte
- Primer:

```
int k=1;
constexpr int duplo(int x) {return 2*x;}
constexpr int a=duplo(1);  // a==2
constexpr int b=duplo(k);  // ! GRESKA
int c=duplo(k);            // c==2
```

Znakovne konstante

- U jeziku C su tipa `int`
- U jeziku C++ su tipa `char`
  - u jeziku C su ‘A’ i 65 ista konstanta,
    a u C++ samo imaju istu numeričku vrednost
    (ako se koristi ASCII skup znakova)
  - iako su konstante tipizirane, u izrazima se mogu zamenjivati:

```
int i='A';
char c=65;
```

Logički tip podataka

- Ključna reč `bool` za uvođenje podataka logičkog tipa
- Logički tip je u grupi celobrojnih tipova
- Vednosti: `true` i `false`
- U izrazima:
  - `false` se konvertuje u `0`,
  - `true` se konvertuje u `1`
- Pri dodeli vrednosti logičkoj promenljivoj:
  - `0` se konvertuje u `false`,
  - nenulta vrednost u `true`
- Relacioni operatori imaju rezultat tipa `bool`
- Logički operatori imaju i operande i rezultat tipa `bool`

Prostori imena

- Mehanizam za izbegavanje konflikta imena
- Prostor imena grupiše globalna imena
- Uvodi se na sledeći način:
  `namespace identifikator { sadržaj }`
- Sadržaj: globalni podaci, funkcije, tipovi i ugnežđeni prostori
- Ime mora biti jednoznačno u datom prostoru
- Ime u prostoru nije u konfliktu sa istim imenom u drugom prostoru
- Pristup imenu `i` iz nekog prostora A se postiže na 3 načina:
  - korišćenjem složenog imena (razrešenjem dosega): `A::i`
  - uvozom datog imena: `using A::i;`
  - uvozom svih imena iz prostora: `using namespace A;`

Niske

- U jeziku C: nizovi znakova (`char[]`, `char*`) koji se završavaju sa `\0`
  - biblliotečke funkcije iz `<string.h>` za rad sa niskama
- U biblioteci jezika C++: tip `string`
- Podrazumevana vrednost objekta tipa `string` je prazna niska `""`
- Sa njim se može raditi slično kao sa primitivnim tipovima:
  - = (dodela), + (spajanje), += (nadovezivanje)
  - relacioni operatori za leksikografsko poređenje
- Automatska konverzija C-niski (`char*`, `"literal"`) u `string`
  - desni operand = i += može biti C-niska
- Literal C++ niske je `const char*`, za razliku od literala C-niske (`char*`)
- Za rad sa niskama – zaglavlje `<string>`, razlicito od `<string.h>`

Niske - primer

- Većina imena iz C++ biblioteke se nalaze u prostoru `std`
- Puno ime tipa C++ niske je `std::string`
- Primer rada sa niskama na jeziku C++:

```
#include <string>
using namespace std; //ukljucivanje imena iz std
string f(string a){
  string b = "Pozdrav!";
  if (a=="***") b=a+" "+b;
  return b;
}
string x=f("");
```

Tipovi `enum`, `struct` i `union`

- Identifikatori nabrajanja, struktura i unija mogu da se koriste kao oznaka tipa
  - nije potrebna ključna reč `enum`, `struct`, `union`

```
enum RadniDan{Pon, Uto, Sre, Cet, Pet};
RadniDan rdan=Pon;
```

- Ako u dosegu postoji objekat (promenljiva) sa istim identifikatorom, sam identifikator označava objekat, a ne tip
  - tada se oznaka tipa piše kao i na jeziku C, sa odgovarajućom ključnom reči `enum`, `struct`, `union`

```
enum OsnovnaBoja{Crvena, Zelena, Plava};
int OsnovnaBoja;
enum OsnovnaBoja b1=Plava;
OsnovnaBoja b2=Zelena; // ! GRESKA
```

Tip nabrajanja

- Svako nabrajanje na jeziku C++ je poseban celobrojni tip
- Tip nabrajanja definiše niz simboličkih konstanti
- Za podatke tipa nabrajanja definisana je samo operacija dodele vrednosti nabrajanja promenljivoj istog tipa
  - eksplicitna konverzija celobrojne vrednosti u tip nabrajanja je obavezna
  - greška ako konvertovana vrednost nema odgovarajuću konstantu u nabrajanju
- Pri korišćenju objekata tipa nabrajanja u aritmetičkim i relacijskim izrazima, automatski se vrši konverzija u `int`

```
enum Dani { PO=1,UT,SR,CE,PE,SU,NE };
Dani dan=SR;
Dani d=3;            // ! GRESKA, obavezna eksplicitna konv.
dan++;               // ! GRESKA, nije definisana operacija ++
dan=(Dani)(dan+1);   // implicitna promocija dan u int,
                     // pa eksplicitna konverzija rezultata
if (dan<NE) {        // dozvoljeno je i: dan<7
```

Pripadajući tip nabrajanja

- Tip koji se koristi za numeričku reprezentaciju nabrajanja
- Podrazumeva se celobrojni tip `int`
- Može se definisati i drugačiji (celobrojni) pripadajući tip:
  `enum ime : pripadajuci_tip {imenovane_konstante}`
- Razlog – kompaktnije zapisivanje vrednosti tipa nabrajanja
- Greške:
  - navođenje vrednosti imenovane konstante izvan opsega pripadajućeg tipa
    ```
    enum Broj :char {NAJMANJI=1, NAJVECI=1000}  // ! GRESKA
    ```
  - prilikom dodele vrednosti nabrajanju, navođenje vrednosti u opsegu pripadajućeg tipa, ali izvan opsega postojećih vrednosti imenovanih konstanti
    ```
    enum Dani : char {PO,UT,SR,CE,PE,SU,NE} ;
    Dani dan=(Dani) 10;  // ! GRESKA – ne otkriva prevodilac
    ```

Nabrajanja sa ograničenim dosegom

- Imenovane konstante nabrajanja imaju isti doseg kao i imena odgovarajućih tipova nabrajanja
- Konflikt imena:

```
enum SemaforPesaci {CRVENO, ZELENO};
enum SemaforVozila {ZELENO, ZUTO, CRVENO}; // ! GRESKA
```

- Rešenje – modifikator `struct` ili `class` iza `enum`:

```
enum struct SemaforPesaci {CRVENO, ZELENO};
enum struct SemaforVozila {ZELENO, ZUTO, CRVENO};
```

- Za korišćenje imenovane konstante potrebno razrešenje dosega:

```
SemaforPesaci semP = SemaforPesaci::CRVENO;
SemaforVozila semV = ZUTO; // ! GRESKA – nije u dosegu
```

- Obavezna eksplicitna konverzija u ceo broj:

```
int i = (int)SemaforVozila::ZUTO;
```

Inicijalizatorske liste

- Lista vrednosti u zagradama: `{vrednost, vrednost,…, vrednost}`
- Mogu da se koriste za inicijalizaciju svih vrsta podataka
  - čak i za proste tipove, kada sadrže samo jednu `vrednost`
  - greška je ako na neku vrednost treba primeniti nebezbednu konverziju tipa
- Vrednosti se dodeljuju redom elementima niza, odnosno poljima strukture
  - za unije – tako može da se postavi samo prvo polje
- Manjak vrednosti dopunjuje se nulama, a višak je greška
- Inic. lista je bezimeni podatak čiji tipovi vrednosti zavise od okruženja
- Mogu da se koriste i pri dodeli vrednosti, osim za nizove
- Argumenti funkcija i izrazi u naredbama `return` mogu biti ovakve liste

Primer inicijalizatorske liste

```
int i1={1}, i2{1}, i3={i1+i2};
i1={2};
int i4={0.5}; // ! GRESKA – nije bezbedno
int *pi={&i1};
int n1[5]={1,2,3}, n2[5]{1,2,3}, n3[]{1,2,3};
int m[][3]{{1,2},{},{1,2,3}};
n1={4,5,6}; // ! GRESKA
struct S1{int a,b;};
S1 s11={1,2}, s12{1,2}; s11={3,4};
struct S2{int a; S1 b; int c[3];};
S2 s21={1,{2,3}, {4,5,6}}, s22{1,2,3,4,5,6};
s21 = {6, {5,4}, {3,2,1}};
```

Bezimena unija

- Unija bez imena predstavlja jedan objekat koji sadrži u raznim trenucima podatke različitih tipova
- Identifikatori članova imaju datotečki ili blokovski doseg, a ne strukturni kao kod unije sa imenom
- Članovi bezimenih unija:
  - koriste se kao obične promenljive

```
union{ int i; double d; char *pc; };
i=55; d=123.456; pc="ABC";
```

- Unija za koju je definisan barem jedan objekat ili pokazivač, ne smatra se bezimenom iako nema ime

Uvek promenljiva polja

- Polje strukture može biti označeno kao mutable
- Takvo polje može da se menja čak i za konstantan objekat strukture
- Primer:

```
struct X{
  int a;
  mutable int b;
};

void main(){
  X x1;
  const X x2;
  x1.a=1;
  x1.b=2;
  x2.a=3;       // ! GRESKA
  x2.b=4;
}
```

Dinamički objekti

- Operator `new` kreira jedan dinamički objekat nekog tipa T
- Operator `delete` uništava dinamički objekat nekog tipa T
- Operand operatora `new` je identifikator tipa T
  sa eventualnim inicijalizatorima (argumentima konstruktora)
- Operator `new`:
  - alocira potreban prostor u memoriji za objekat datog tipa
  - zatim poziva konstruktor tipa
- Ako nema dovoljno prostora – izuzetak `bad_alloc` (zaglavlje `<new>`)
  - pre standarda – rezultat `nullptr` (ranije `NULL` ili `0`)
- Može se izbeći bacanje izuzetka: `X *x = new (nothrow)X;`
- Operator `new` vraća pokazivač na dati tip:

```
int *pi = new int;
Tacka *pt1 = new Tacka(5.0,5.0);
```

- Dinamički objekat nastaje kada se izvrši operacija `new`,
  a traje sve dok se ne izvrši operacija `delete`

Uništavanje dinamičkih objekata

- Operator `delete` ima jedan argument tipa pokazivača nekog tipa
- Ovaj pokazivač mora da ukazuje na objekat kreiran pomoću `new`
  - ako pokazivač ne ukazuje na objekat kreiran pomoću `new`, posledice `delete` su nepredvidive
  - ako je pokazivač `nullptr`, `delete` samo nema efekta
- Operator `delete`
  - poziva destruktor za objekat na koji ukazuje pokazivač
  - zatim oslobađa zauzeti prostor
- Operator `delete` vraća `void` (bez rezultata)

```
Tacka *pt;
void f() { pt=new Tacka(0.1,0.2); }
void main () { f(); … delete pt; }
```

Dinamički nizovi

- Operatorom `new` može se kreirati i niz objekata nekog tipa:

```
Tacka *pt = new Tacka[10];
```

- Sve dimenzije niza osim prve treba da budu konstantni izrazi
  - prva dimenzija može da bude i promenljiv izraz
  - promenljiv izraz mora biti takav da može da se izračuna u trenutku izvršavanja naredbe sa operatorom `new`
- Inicijalizacija objekata elemenata niza
  - podrazumevanim konstruktorom
  - ako ga klasa nema - prevodilac ga automatski generiše
- Dinamički niz se ukida operatorom `delete` sa parom zagrada:

```
delete [] pt;
```

- Redosled konstrukcije elemenata je po rastućem indeksu
- Redosled destrukcije je obrnut od redosleda konstrukcije

Reference (upućivači)

- U jeziku C prenos argumenata u funkciju - isključivo po vrednosti (*by value*)
- Da bi funkcija mogla da promeni vrednost spoljne promenljive, trebalo je preneti pokazivač na tu promenljivu, pa indirektno adresirati u f-ji
- C++ uvodi izvedeni tip *reference* (*upućivača*) na objekat
- U jeziku C++ moguć je i prenos argumenta po adresi (*by reference*)

```
void f(int i, int &j){ // i po vrednosti, j po referenci
  i++;                 // stvarni argument se neće promeniti
  j++;                 // stvarni argument će se promeniti
}
void main () {
  int si=0,sj=0;
  f(si,sj);
  cout<<"si="<<si<<", sj="<<sj<<endl;
}
Izlaz: si=0, sj=1
```

Definisanje referenci

- Reč je o referencama na *lvrednosti* (podrazumevano značenje “referenca”)
- Reference se deklarišu upotrebom znaka & ispred imena
- Referenca je alternativno ime za neki objekat (alias, sinonim)
- U definiciji referenca mora da se inicijalizuje objektom na koga će upućivati
- Od inicijalizacije referenca postaje sinonim za objekat na koga upućuje
- Svaka operacija nad referencom (uključujući i operaciju dodele) je operacija nad objektom na koji referenca upućuje

```
int i=1;   // celobrojni objekat i
int &j=i;  // j upućuje na i
i=3;       // menja se i
j=5;       // opet se menja i
int *p=&j; // isto što i &i
j+=1;      // isto što i i+=1
int k=j;   // posredan pristup do i preko reference
int m=*p;  // posredan pristup do i preko pokazivača
```

Implementacija referenci

- Referenca je slična konstantnom pokazivaču na objekat datog tipa
- Referenca pri inicijalizaciji dobija vrednost adrese objekta kojim se inicijalizuje
- Nema načina da se, posle inicijalizacije, vrednost reference promeni
- Svako obraćanje referenci podrazumeva posredni pristup objektu, preko reference
- Posredan pristup preko pokazivača se vrši operatorom *,
  a preko reference bez posebne operacije
- Uzimanje adrese (operator &) reference vraća adresu objekta na koji ona upućuje

```
int &j = *new int(2);  // j upućuje na dinamički objekat 2
int *p=&j;             // p je pokazivač na isti objekat
(*p)++;                // objekat postaje 3
j++;                   // objekat postaje 4
delete &j;             // isto kao i delete p
```

- Ako je referenca tipa reference na nepromenljiv podatak,
  objekat na koji ona upućuje se ne sme promeniti preko te reference
- Ne postoje nizovi referenci, pokazivači na reference, ni reference na reference
- Referenca na pokazivač je dozvoljena, npr.

```
int i=5,*p=&i,*&rp=p;
```

Funkcije koje vraćaju referencu

- Referenca (na *lvrednost*) može i da se vrati kao rezultat funkcije
- U tom slučaju funkcija treba da vrati referencu na objekat koji traje (živi) i posle izlaska iz funkcije.
- Primer:

```
int& f(int &i){int &r=*new int(i); return r; }// OK
int& f(int &i){return *new int(i); }           // OK
int& f(int &i){return i; }                     // OK

int& f(int &i){int r=i;  return r; }          // ! GRESKA
int& f(int i){return i; }                     // ! GRESKA
int& f(int &i){int r=*new int(i); return r; }// ! GRESKA
int& f(int &i){int j=i, &r=j; return r; }     // ! GRESKA
```

- Rezultat poziva funkcije je *lvrednost* samo ako funkcija vraća referencu

Obilazak elemenata niza u petlji

- Nova vrsta petlje `for` koja služi za sistematski obilazak elemenata niza:
  `for (tip prom: niz) naredba`
- `prom` u svakom ciklusu petlje sadrži kopiju narednog elementa niza
- Promena `prom` ne utiče na element niza
- Ako se navede: `for (tip &prom: niz) naredba`
  - tada je `prom` referenca na tekući element niza koji tako može da se promeni
- Primer:

```
int  niz[5] = {1,2,3,4,5};
s=0; for (int k: niz) s+=k; cout<<s<<endl; // 15
for (int &k: niz) k*=2;
for (int k: niz) cout<<k<<‘ ‘; cout<<endl; // 2 4 6 8 10
```

Reference na *dvrednosti* (1)

- *Lvrednosti* su izrazi koji upućuju na nešto u memoriji, čemu može da se dohvati adresa
- *Dvrednosti* su izrazi koji nisu lvrednosti:
  - literali, aritmetički, relacijski, logički i adresni izrazi
- Tip reference na *dvrednost*: `osnovni_tip &&`
- Mogu da upućuju samo na *dvrednosti* (promenljive i nepromenljive)
- Referenca na dvrednost je *lvrednost* !
- Posledica: privremeni podaci (rezultati izračunavanja) dobijaju imena
  - preko tog imena mogu da se promene privremeni podaci
- Primer:

```
int i=1;         // promenljiva vrednost
const int ci=i; // nepromenljiva vrednost
int && rd1=i;   // ! GRESKA – (promenljiva) lvrednost
int && rd2=ci;  // ! GRESKA – (nepromenljiva) lvrednost
int && rd3=i+1; // promenljiva dvrednost
int && rd4=10;  // nepromenljiva dvrednost
rd3++; rd4++;   // rd3==3, rd4==11
```

Reference na *dvrednosti* (2)

- Podatak na koji upućuje referenca na *dvrednost* može biti nepromenljiv
  - tada ne mogu da se menjaju privremeni podaci na koje upućuje referenca
- Primer

```
int i=1;          // promenljiva vrednost
const int ci=i; // nepromenljiva vrednost
const int && crd1=i;        // ! GRESKA – lvrednost
const int && crd2=ci;       // ! GRESKA – lvrednost
const int && crd3=i+1;      // promenljiva dvrednost
const int && crd4=10;       // nepromenljiva dvrednost
crd3++; crd4++; // ! GRESKA – ref. na nepromenljive vr.
```

Reference na *dvrednosti* kao parametri

- Parametri funkcije mogu biti tipa reference na *dvrednosti*
- U tom slučaju stvarni argument može biti izraz koji je *dvrednost*
- Po potrebi se primenjuje i konverzija tipa argumenta u tip parametra
- Za parametre tipa ref. na *dvrednost* ne postoji bočni efekat
  - zato za njih modifikator `const` nema smisla
  - takav parametar je privremeni podatak koji nije vidljiv van funkcije
  - i kad je argument *lvrednost*, od nje se pravi privremeni podatak
- Primer:

```
void povecaj(int&& a) {a++;}
int x=1;
povecaj(x);   // x je lvrednost; po povratku: x==1
povecaj(x+1); // x+1 je dvrednost; pri izlasku: a==3
```

Neposredno ugrađivanje funkcija (1)

- Često se definišu vrlo jednostavne, kratke funkcije
  - na primer, neke samo prosleđuju argumente drugim funkcijama
  - vreme koje se troši na prenos argumenata i poziv može biti veće nego vreme izvršavanja tela same funkcije
- Ovakve funkcije se mogu definisati uz zahtev da se neposredno ugrađuju u kôd (*inline* funkcije)
  - telo takve funkcije direktno se ugrađuje u kôd na mestu poziva funkcije
  - semantika poziva ostaje potpuno ista kao i za običnu funkciju
- Ovakva funkcija definiše se dodavanjem specifikatora `inline` :

```
inline int inc(int i) {return i+1;}
```

Neposredno ugrađivanje funkcija (2)

- Funkcija članica klase je ugrađena
  ako se definiše unutar definicije klase
- Ako se definiše izvan definicije klase, funkcija je ugrađena
  kada se ispred njene definicije nalazi reč `inline`:

```
class C {
  int i;
public:
  int val ()
    {return i;}
};
```

```
class D {
  int i;
public:
  int val();

};
inline int D::val()
  {return i;}
```

Neposredno ugrađivanje funkcija (3)

- Prevodilac ne mora da poštuje zahtev za neposredno ugrađivanje
  - za korisnika ovo ne predstavlja problem, jer je semantika ista
  - ugrađene funkcije samo mogu da ubrzaju program, a nikako da izmene njegovo izvršavanje
- Ako se ugrađena funkcija koristi u više datoteka, u svakoj datoteci mora da se nađe njena potpuna definicija
  - ovo je najbolje sprovesti pomoću datoteke-zaglavlja u kojoj je definicija funkcije za ugrađivanje
  - nedostatak: *.h datoteka ne sadrži samo interfejsne već i implementacione elemente (otkriva se “poslovna tajna”)
- Ugrađene funkcije eliminišu potrebu za makroima:
  `#define` <*ime*>`(`<*lista argumenata*>`)` <*tekst zamene*>

Ugrađene funkcije umesto makroa

- Makrodefinicija:

```
#define max(i,j)((i)>(j))?(i):(j)
```

- Makropoziv:

```
max(k++,l++)
```

- Makroekspanzija:

```
((k++)>(l++))?(k++):(l++)
```

- Problem: jedan argument se 2x inkrementira
- Ugrađena f-ja:

```
inline int max(int i, int j)
        {return i>j?i:j;}
```

- Ne postoji gornji problem

Podrazumevane vrednosti argumenata

- C++ podržava podrazumevane vrednosti argumenata u definiciji funkcije
- Formalni argument uzima podrazumevanu vrednost
  ako se pri pozivu funkcije ne navede odgovarajući stvarni argument

```
Kompleksni::Kompleksni(float r=0, float i=0)
  {real=r; imag=i;}
void main () {
  Kompleksni c1, c2(0), c3(0,0); // sva tri objekta (0,0)
}
```

- Podrazumevane vrednosti su proizvoljni izrazi
  - izračunavaju se svaki put pri pozivu funkcije
- Podrazumevani argumenti mogu da budu samo nekoliko poslednjih iz liste:

```
Kompleksni::Kompleksni(float r=0,float i) // ! GRESKA
  {real=r;imag=i;}
Kompleksni::Kompleksni(float r,float i=0) // Ispravno;
  {real=r;imag=i;}
void main(){Kompleksni c(0);}
```

Preklapanje imena funkcija

- Često su potrebne funkcije koje realizuju logički istu operaciju, samo sa različitim tipovima argumenata
- Za svaki od tih tipova mora da se realizuje posebna funkcija
- U jeziku C to bi moralo da se realizuje tako da te funkcije imaju različita imena, što smanjuje čitljivost programa
- U jeziku C++ moguće je definisati više različitih funkcija sa istim identifikatorom
- Ovakav koncept naziva se *preklapanje imena funkcija* (engl. *name overloading*)
- Uslov je da im se razlikuje broj i/ili tipovi argumenata, odnosno potpis funkcije
- Tipovi rezultata ne moraju da se razlikuju i nije dovoljno da se samo oni razlikuju:

```
double max (double i, double j)
  { return (i>j) ? i : j; }
char* max (const char *p, const char *q)
  { return (strcmp(p,q)>=0)?p:q; }
double r=max(1.5,2.5);       // max(double,double)
double r=max(1,2.5);         // (double)1; max(double,double)
char *q=max("Pera","Mika"); // max(const char*,const char*)
```

Razlučivanje kod preklapanja imena

- Koja će se funkcija stvarno pozvati, određuje se u fazi prevođenja
  - mehanizam je potpuno statički
- Određivanje se vrši prema slaganju potpisa funkcija
  - uspoređuje se broj i/ili tipovi stvarnih i formalnih argumenata
- Potrebno da prevodilac može jednoznačno da odredi koja funkcija se poziva
- Pravila za određivanje koja funkcija se poziva su veoma složena
- U praksi se svode samo na dovoljno razlikovanje potpisa funkcija
- Prevodilac približno ovako prioritira slaganje tipova argumenata:
  - potpuno slaganje tipova, uključujući trivijalne konverzije (trivijalne konverzije su npr. iz tipa `T[]` u tip `T*`, ili iz `T` u `T&` i obrnuto)
  - slaganje tipova korišćenjem standardnih konverzija (npr. `char` u `int`)
  - slaganje tipova korišćenjem korisničkih konverzija

Brisanje funkcija

- Sprečavanje da se pozove za neki tip argumenta
  - za koji bi inače mogla da se pozove konverzijom tipa argumenta
- Preklapa se ime funkcije:
  - navodi se potpis funkcije sa parametrima neželjenog tipa
  - dodaje se `=delete` iza liste parametara
- Prevodilac otkriva pokušaj poziva obrisanog preklapanja funkcije
- Primer:

```
void f(double x){}
void f(int x) =delete;
f(0.5);    // u redu
f(1);      // ! GRESKA
f('a');    // ! GRESKA – izbor f-je nije jednoznacan
```

Pristup elementima (1)

- Reč je o elementima složenih podataka kakvi su nizovi ili strukture
- Pristup elementima promenljivog i nepromenljivog niza:

```
      int& elem(      int *a, int i) {return a[i];}
const int& elem(const int *a, int i) {return a[i];}
```

- Prva funkcija omogućava menjanje elementa niza preko svog rezultata
- Druga funkcija sprečava promenu i u funkciji i preko rezultata
- Primer:

```
int a[20],i=10;
const int b[20]={0};
elem(a,i)=1;
elem(b,i)=1; // !GRESKA
int x=elem(b,i);
```

Pristup elementima (2)

- Problem može biti što su opisane dve funkcije sličnog sadržaja
  - to izaziva održavanje dva “klona” tela funkcije, što je sklono greškama
  - može se napisati druga funkcija koja poziva prvu

```
const int& elem(const int *a, int i)
    {return elem(const_cast<int*>(a),i);}
```

- Za funkcije koje vraćaju pokazivač

```
      int* adrEl(      int *a, int i) {return a+i;}
const int* adrEl(const int *a, int i) {return a+i;}
*adrEl(a,i)=2;
*adrEl(b,i)=2; // ! GRESKA
int y= *adrEl(b,i);
```

- Slično važi za polja struktura

Napomene i funkcije koje se ne vraćaju

- Napomene (atributi za prevodioca, anotacije):
  - navode se u obliku `[[napomena]]`
  - mogu da se stave uz razne elemente: funkcije, parametre, promenljive,…
  - služe prevodiocu za provere i optimizaciju
  - ne utiču na ponašanje logički ispravnog programa
  - prevodioci mogu da ih zanemare
- Neke funkcije se ne vraćaju na mesto poziva
  - razlog može biti što takve funkcije “nasilno” prekidaju rad programa
  - rad programa se prekida sa `exit(parametar)`
  - parametar izlaska iz programa može biti `0` (OK) ili indikator greške `!=0`
  - napomena da se funkcija ne vraća na mesto poziva: `[[noreturn]]`
  - ako u funkciji postoji `return`, posledice su nepredvidive (zavisi od prevodioca)

Operatori i izrazi

- Novi operatori (12):
  - `::`, `new`, `delete`, `.*`, `->*`, `typeid`, `throw`, `alignof`, *cast* operatori (4)
- Postfiksne varijante `++` i `--` imaju viši prioritet
- Podsećanje:
  - rezultat prefiksnih operatora `++` i `--` je *lvrednost*
  - rezultat operatora dodele vrednosti je *lvrednost*
  - rezultat uslovnog izraza je *lvrednost* ako su drugi i treći operand *lvrednosti* istog tipa

```
int x,a=0,b=0;
x?a:b=1; //if(x) a=1; else b=1;
```

Pregled operatora (1)

Prior.	Br. op.	Asoc.	Operator
10	2	→	`== !=`
9	2	→	`&`
8	2	→	`^`
7	2	→	`|`
6	2	→	`&&`
5	2	→	`||`
4	3		`?:`
3	2	←	`= += -= *= /= %= &= ^= |= <<= >>=`
2	1		`throw`
1	2	→	`,`

Pregled operatora (2)

Prior.	Br. op.	Asoc.	Operator
19	1,2	→	`::`
18	2	→	`[] () . ->`
17	1		`a++ a-- specificni_cast(4) typeid`
16	1	←	`! ~ ++a --a - + * & (tip) sizeof alignof new delete`
15	2	→	`.* ->*`
14	2	→	`* / %`
13	2	→	`+ -`
12	2	→	`<< >>`
11	2	→	`< <= > >=`

Operatori konverzije tipa

- U jeziku C postoji operator za konverziju tipa (engl. *cast*):
  `(`*`tip`*`)`*`izraz`*
- Kast operator iz jezika C se može koristiti, ali se ne preporučuje
- U jeziku C++ uvode se dodatni specifični kast operatori:
  `static_cast <`*`oznaka_tipa`*`> (`*`izraz`*`)`
  `reinterpret_cast <`*`oznaka_tipa`*`> (`*`izraz`*`)`
  `const_cast <`*`oznaka_tipa`*`> (`*`izraz`*`)`
  `dynamic_cast <`*`tip_pokazivača_ili_reference`*`> (`*`izraz`*`)`
- Neke konverzije tipa su bezbedne, a neke nisu
  - npr. `int` → `float` jeste bezbedna, ali `float` → `int` je rizična
- Po pravilu, za nebezbedne konverzije zahteva se eksplicitan kast
- Notacija je nezgrapna iz dva razloga
  - da se lakše može uočiti u tekstu programa
  - da se programeri odvraćaju od korišćenja
- Potreba za eksplicitnom konverzijom – signal za preispitivanje projektnih odluka

Statička konverzija

- **Statički kast** je namenjen uglavnom za prenosive konverzije, npr.:
  - između numeričkih tipova (računajući i tip nabrajanja `enum`)
  - između pokazivača proizvoljnih tipova i `void*`
  - nestandardne konverzije (koje definiše programer)
- Ove konverzije se primenjuju automatski (implicitno) kada su bezbedne, a eksplicitna konverzija je porebna za nebezbedne konverzije, npr.:
  - konverziju pokazivača `void*` u pokazivač na poznati tip
  - konverziju numeričkog tipa u tip nabrajanja (`enum`)
- Primeri:

```
float a = 5.0; int i = static_cast<int>(a);
        // kao: int i = (int)a;
void *p = static_cast<void *>(&i); // moze: void* p = &i;
int *q = static_cast<int *>(p); // mora eksplicitni kast
```

- U C++ uveden literal pokazivača `nullptr` (tipa `void*`)
  - može da se (bezbedno, bez kasta) dodeljuje svim tipovima pokazivača
  - ne preporučuje se više korišćenje: `0` ni `NULL` iz `<stddef.h>`

Reinterpretirajuća konverzija

- **Reinterpretirajući kast** je namenjen konverziji tipova bez logičke veze, npr:
  - između celobrojnih vrednosti i pokazivača
  - između pokazivača ili referenci na razne tipove
- Suština je da nema pretvaranja vrednosti već se samo ista vrednost različito interpretira
- Konverzija nije bezbedna i malo je verovatno da je prenosiva
- Primeri:

```
int i = 1155;
short *p = reinterpret_cast<short *>(i);
int j = reinterpret_cast<int>(p); // j==i
float *q = reinterpret_cast<float *>(&i);
float &r = reinterpret_cast<float &>(i);
```

Konstantna konverzija

- **Konstantni kast** je namenjen uklanjanju ili dodavanju **`const`**
- Dodavanje specifikatora je bezbedno, ali uklanjanje nije
  - jer omogućava promenu vrednosti nepromenljivog podatka
- Primeri:

```
int j = 1;
const int i = j;
int *p = const_cast<int *>(&i); // mora eksplicitno
*p = 0;          // promena “nepromenljivog” podatka
double pi = 3.14;
const double &cpi = const_cast<const double &>(pi);
                          // ili: const double &cpi=pi;
pi = 0.0;                 // OK
cpi = 0.0;                // ! GRESKA
```