OBJEKT-ORIJENTISANO PROGRAMIRANJE

Oznaka predmeta: OOP

Predavanje broj: 5

Nastavna jedinica: Tipovi podataka.

Nastavne teme: Pokazivači. Nizovi i pokazivači. Nizovi znakova i pokazivači na znakove. Strukture. Unije. Bit-polja.

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Dragan Milićev, “Objektno orijentisano programiranje na jeziku C++”, Mikro knjiga, Beograd, 2005.

Pokazivači

- Objekti tipa pokazivača na tip T deklarišu se kao: T **ime*;
  - Tip T ne može biti referenca ili bit polje.
- Pristup objektu na koji ukazuje pokazivač p (čiju adresu sadrži) vrši se operacijom dereferenciranja (operator *).
- Ako pokazivač p ukazuje na neki objekat u nizu a, onda je definisan rezultat operacije sabiranja tog pokazivača p sa nekom celobrojnom veličinom i.
- Rezultat ove operacije sabiranja je pokazivač istog tipa kao i tip pokazivača-operanda p. Pokazivač-rezultat ukazuje na objekat istog niza a koji je udaljen za odgovarajući ceo broj mesta od objekta na koga pokazivač-operand p ukazuje.
  - Analogno važi za operaciju inkrementiranja pokazivača (operator ++) i operaciju sabiranja i dodele (operator +=). Na primer:

```
int a[10];       // niz a sa 10 elemenata tipa int
int *pi=&a[0];   // pi je pokazivač na int koji ukazuje na a[0]
*(pi+3)=4;       // pi+3 je pokazivač koji ukazuje na a[3],
                 // pa a[3] postaje 4
pi++;     // pi sada ukazuje na naredni element niza a, tj. a[1]
```

Pokazivači

- Rezultat sabiranja pokazivača sa celobrojnom veličinom ukazuje na lokaciju u memoriji koja je udaljena od mesta gde pokazuje pokazivač za vrednost proizvoda te celobrojne veličine i veličine objekta na koji ukazuje pokazivač.
  - Ako rezultat sabiranja pokazivača i celobrojne veličine ukazuje na lokaciju van niza neće doći do greške u prevođenju jer se vrednost pokazivača zna tek u vreme izvršavanja programa.
  - Kako veličina objekta može da varira zavisno od platforme, na ovo bi trebalo obratiti posebnu pažju ako se želi prenosivost programa.
- Garancija da je rezultat sabiranja definisan i kada ukazuje na jedno mesto iza gornje granice niza omogućava pisanje petlji unutar kojih se sekvencijalno pristupa elementima nekog niza preko pokazivača, kada taj pokazivač u poslednjoj iteraciji prekoračuje gornju granicu niza.
  - Nije definisan rezultat operacije dereferenciranja (*) takvog pokazivača, iako većina prevodilaca to ne prijavljuje kao grešku ni u vreme izvršavanja.

```
void f(int a[], int n){//f-ja f obrađuje svih n elemenata niza a
     int *p=&a[0];
     for(int i=0; i<n; i++){
       // ... radi nešto sa *p
       p++;       // u poslednjoj iteraciji p prekoračuje granicu
     }}
```

Pokazivači

- Razultata oduzimanja celog broja od pokazivača trebalo bi da ukazuje na element niza i da pokazivač-rezultat ukazuje na element niza unutar njegovih granica.
  - Pokazivač-rezultat ukazuje na element niza za odgovarajući ceo broj mesta ispred elementa na koji ukazuje pokazivač-operand.
  - Oduzimanje je definisano i za pokazivač koji za jedno mesto prekoračuje granicu niza
- Kod oduzimanje dva pokazivača istog tipa oba pokazivača treba da ukazuju na elemente istog niza. Rezultat je ceo broj koji predstavlja rastojanje u broju elemenata između elemenata na koje ukazuju pokazivači-operandi.
  - ako pokazivači ukazuju na susedne elemente niza, njihova razlika je 1.

```
int a[10];
int *p1=&a[0], *pi2=&a[4];
int i=pi2-pi1;                  // i je jednako 4
pi2-=i;                  // pi2=pi2-4, pi2 sada ukazuje na a[0]
```

- Kod pokazivača na funkcije odgovarajućeg tipa tip pokazivača je "pokazivač na funkciju koja ima argumente tog-i-tog tipa i vraća rezulta tog-i-tog tipa".
  - Samo pokazivači na funkcije istog tipa (koje imaju isti broj i tipove argumenata, kao i isti tip rezultata) mogu se direktno dodeljivati jedan drugome.

Pokazivači

```
int f(int);     // f je funkcija koja ima jedan argument tipa int i
                // vraća rezultat tipa int
int g();        // g je funkcija koja nema argumente i
                // vraća rezultat tipa int
void h(int)     // h je funkcija koja ima jedan argument tipa int i
                // ne vraća rezultat
int (*pf)(int);// pf je pokazivač na funkciju koja ima jedan
                // argument tipa int i vraća rezultat tipa int
pf=&f;          // pf ukazuje na funkciju f
pf=&g;          // greška!
pf=&h;          // greška!
```

- U prethodnom primeru treba obratiti pažnju na način na koji je pokazivač na funkciju ***pf*** deklarisan: `int (*pf)(int);`
- Prvu dodelu vrednosti pokazivaču ***pf*** treba tumačiti na sledeći način.
  - Na desnoj strani znaka = nalazi se izraz operacije & nad funkcijom f tako da je rezultat objekat tipa "pokazivač na funkciju koja ima jedan argument tipa int i vraća rezultat tipa int" (tip "pokazivač na tip argumenta operacije &").
- Kod druge dodele, rezultat operacije & nad funkcijom g je tipa "pokazivač na funkciju bez argumenata koja vraća tip int". Ovaj rezultat je drugog tipa od pf, pa mu se ne može dodeliti. Slično važi i za treću dodelu.

Pokazivači

- Rezultat operacije dereferenciranja (*) pokazivača na funkciju je funkcija na koju on ukazuje, pa se sa tim rezultatom može raditi samo ono što se sa tom funkcijom može raditi, npr. može se pozvati ta funkcija.

  Primeri:

```
int f(int);         // f, g i h su funkcije koje
int g(int);         // imaju jedan argument tipa int
int h(int);         // i vraćaju rezultat tipa int;
int (*pf)(int);     // pf je pokazivač na takvu funkciju;

pf=&f;            // pf ukazuje na f, ili pf=f
int i=(*pf)(7);// poziva se funkcija na koju ukazuje pf
                  // sa argumentom 7 i rezultat smešta u i;
pf=&g;            // pf sada ukazuje na g, ili pf=g
i=(*pf)(3);       // poziva se funkcija na koju ukazuje pf
                  // sa argumentom 3;
pf=&h;            // pf sada ukazuje na h, ili pf=h

*pf=f;            // greška!
```

Nizovi i pokazivači

- Ova veza sastoji se u tome što se uvek kada se identifikator niza nađe u nekom izrazu (osim sa operatorima sizeof i &, kao i pri inicijalizaciji referenci), implicitno konvertuje u objekat tipa pokazivača na tip svojih elemenata.
  - Vrednost tog pokazivača je takva da on ukazuje na prvi element tog niza.
  - Nizovi nisu promenljive lvrednosti.
- Za pristup elementu niza, prevodilac treba da zna dimenzije niza i početnu adresu tog niza.
  - Za smeštanje početne adrese niza služi tip pokazivača na element niza.
  - Dimenzija niza koristi se u fazi prevođenja samo za alociranje memorijskog prostora za niz, kao i kod pristupa elementima višedimenzionih nizova.
    - ključne informacije koje prevodilac treba da ima kada se koristi niz je tip njegovih elemenata i početna adresa.
- Identifikator niza se može naći na potpuno proizvoljan način u izrazu

```
int a[10], b[10];  // nizovi a i b od po 10 elemenata tipa int
a=b;               // greška, niz nije promenljiva lvrednost!
```

  - Fiktivni pokazivači a b nisu (kao rezultat konverzije) nisu lvrednosti, pa je dodela a=b nemoguća. Kopiranje nizova se u jeziku C++mora vršiti element po element.

Nizovi i pokazivači

- Indeksiranje niza je u jeziku C++ operacija koja se označava operatorom [].
- Osim u slučajevima kada korisnik definiše značenje ovog operatora za klase, izraz E1[E2] je potpuno ekvivalentan izrazu *((E1)+(E2)).
  - Kako je E1 najčešće niz od njega se formira fiktivni objekat - pokazivač na prvi element tog niza, a E2 je najčešće celobrojni izraz.
  - Rezultat E1[E2] gleda na E2-ti element niza E1.
  - operacija dereferenciranja (*) daje odgovarajući element niza.

```
int a[10];
int i=0;
a[i]=5;       // isto što i: *(a+i)=5;
(a+1)[i]=3;   // isto što i: *((a+1)+i)=3, pa a[i+1] postaje 3;
int *pi=a;    // a se konvertuje u pokazivač na int,
              // pa pi ukazuje na a[0];
pi[i+2]=1;    // isto što i: *(pi+(i+2))=1, pa a[i+2] postaje 1;
(i+3)[a]=0;   // može i ovo, isto što i: *((i+3)+a)=0, pa a[i+3]
              // postaje 0
```

- Sabiranje je komutativno tako da je E1+E2 isto što i E2+E1, odnosno, operacija indeksiranja ([]) je komutativna!
  Isto pravilo se dosledno primenjuje i kod višedimenzionih nizova.

Nizovi i pokazivači

- Ako je niz a deklarisan sa:

  ```
  int a[5][7];
  ```

  onda se svako pojavljivanje ***a*** u izrazima konvertuje u pokazivač na (prvi od pet) elemenata tipa "niz od 7 elemenata tipa int".

- Na primer, izraz a[i] se tumači kao *(a+i).
  - Pri tome se a konvertuje u pokazivač na (prvi) element tipa "niz od sedam elemenata tipa int".
  - Zatim se ovaj pokazivač sabira sa celim brojem ***i*** da bi se dobio pokazivač na i-ti element niza a.
  - Rezultat operacije dereferenciranja (*) je sam taj element, a on je tipa "niz od sedam elemenata tipa int".
  - Ovaj niz se opet konvertuje u pokazivač na prvi element tipa int.
  - Ako se iza a[i] nalazi još jedna operacija indeksiranja, postupak će se dosledno nastaviti dok se ne dođe do elementa tipa int.

Nizovi znakova i pokazivači na znakove

- String literalima se definišu konstante tipa niza znakova.
  - Ovi nizovi znakova se uvek završavaju znakom '\0'.
  - String literali imaju tip "niz elemenata tipa char", statički su po životnom veku, a pokušaj izmene ovog niza ima nedefinisane posledice.
- Pošto je tip string literala niz znakova a ovaj tip se implicitno konvertuje u tip pokazivača na znak, onda se objekti koji treba da čuvaju nizove znakova formiraju na jedan od sledeća dva načina:

```
char *p="Zdravo!"; char s[7]="Hello!";
```

- Prvu definiciju prevodilac tretira na sledeći način:
  - U oblasti memorije za statičke objekte odvoja prostor potreban za smeštanje stringa "Zdravo!", veličine 8 elemenata tipa char, u njega upisuje odgovarajuće znakove (u poslednji znak upisuje '\0'),
  - Pokazivač p inicijalizuje se tako da ukazuje na prvi element tog niza.
    - preciznije, string literal, koji je tipa "niz znakova", se konvertuje u pokazivač na svoj prvi element, a zatim se dobijenim pokazivačem inicijalizuje pokazivač p.
- Druga definicija prikazuje drugi način definisanja niza znakova uz inicijalizaciju string literalom.
  - niz s sadrži znakove: 'H', 'e', 'l', 'l', 'o', '!' i '\0‘, potrebno je rezervisati mesto za 7 elemenata, sva slova i još znak '\0' koga dodaje prevodilac na kraj niza.

Nizovi znakova i pokazivači na znakove

- Sa nizovima znakova se najčešće i operiše preko pokazivača na znakove.

```
int strlen (char *s) {// izračunava dužinu niza s u znakovima
  int i=0;
  for (; *s++; i++);
  return i;
}
```

- Funkcija strlen ima formalni argument **s** tipa pokazivača na znak, pa se kao stvarni argument može dostaviti objekat istog tipa, kao i tipa niz znakova, jer će se u tom slučaju izvršiti implicitna konverzija u tip pokazivača na znak.
- Petljom **for** se prolazi kroz niz znakova na koji ukazuje **s** i u čijem se svakom koraku odbrojava dužina niza koja se dobija u promenljivoj **i**. Ovde **return** vraća ovako dobijenu vrednost kao rezultat funkcije.
- U izrazu ***s++** , uzima se znak na koji ukazuje **s**, a onda se **s** inkrementira za 1.
  - Ako je znak '\0', petlja se završava, jer se stiglo do poslednjeg znaka u nizu.
  - U suprotnom se izvršava telo petlje (prazno je), a zatim izraz **i++**, kojim se brojač dužine niza samo inkrementira.
  - Pokazivač **s** je inkrementiran, pa pokazuje na sledeći znak u nizu.
    - **s** je kopija stvarnog argumenta (stvarni argument se ne menja).
    - **s** posle poslednje iteracije ukazuje iza poslednjeg znaka u nizu.

Nizovi znakova i pokazivači na znakove

- Funkcija strlen se može pozvati kao:

```
char *s="Zdravo!";
len1=strlen("Kako se zoves?");
len2=strlen(s);
```

- Ovakvu "kratkoću izražavanja" kao u prikazanoj funkciji jezik C++ je nasledio od jezika C.
    - Stil porogramiranja u ovim jezicima je upravo što sažetije pisanje izraza i što intenzivnije korišćenje operatora. Bogatstvo operatora omogućava koncizno pisanje kôda.
- Funkcija strsrch traži znak u nizu znakova i vraća poziciju ovog znaka unutar niza ako ga nađe, inače vraća -1.

```
int strsrch (char *p, char q)
{
  for (int i=0; *p; i++)
    if (*p++ == q) return i;
  return -1;
}
```

- Standardno zaglavlje <cstring> sadrži deklaracije velikog broja bibliotečnih funkcija za rad sa nizovima znakova.

Nizovi znakova i pokazivači na znakove

- Konstantni tip je tip izveden iz nekog drugog tipa.
  - Ovaj tip se označava dodavanjem ključne reči const ispred specifikatora tipa u deklaraciji nekog objekta.
  - Pokušaj izmene vrednosti konstantog objekta prevodilac prijavljuje kao grešku. Konstantni objekti nekog tipa imaju sve druge osobine tog tipa.

```
const float pi  =3.14f;     const char  plus='+';
pi=5; // greška!
```

- Celobrojni konstantni objekat koji je inicijalizovan nekim konstantim izrazom može se upotrebiti u konstantom izrazu.
  - Prevodilac često i ne odvaja poseban prostor za ovakve objekte, već njihovu upotrebu razrešava u fazi prevođenja, ugrađujući njihove vrednosti direktno u kôd kao konstante.

```
const int n=100;
int a[n], b[2*n]; // n je konstanta
```

- Svaki element konstantog niza je konstantan,
  svaki nestatički podatak član konstantog objekta neke klase je konstantan.
- Pokazivač na konstantu deklariše se stavljanjem reči const ispred specifikatora tipa na koji ukazuje pokazivač (ispred cele deklaracije), a konstantni pokazivač deklariše se stavljanjem reči const ispred samog imena pokazivača.

Nizovi znakova i pokazivači na znakove

- Kako konstanti objekat nije promenljiva lvrednost, to ni rezultat operacije dereferenciranja (*) pokazivača na konstantni objekat nije promenljiva lvrednost.

```
const char *pk="asdfgh";          // pokazivač na konstantu
pk[3]='a';                        // greška!
pk="qwerty";                      // ispravno
char c[]="asdfgh";
char *const kp=c;                 // konstantni pokazivač
kp="qwerty";                      // greška!
kp[3]='a';                        // ispravno
const char *const kpk="asdfgh";// konstantni pokazivač na konstantu
kpk[3]='a'; kpk="qwerty";         // greška!
```

- Ako je formalni argument funkcije (pokazivača na) konstantni objekt obezbeđuje se da se taj objekat ne može izmeniti u telu funkcije.

```
int strlen (const char *s);
```

- Ako funkcija ima konstantni povratni tipa privremeni objekat koji prihvata vraćenu vrednost funkcije biće odgovarajućeg konstantnog tipa
  Ako funkcija vraća pokazivač na konstantni objekat taj objekat se ne može dalje menjati posle povratka iz funkcije

```
const int* f();      // f vraća pokazivač na const int
*f()=3;              // greška, jer f vraća const int*!
```

Nizovi znakova i pokazivači na znakove

- Svuda gde se kao operand neke operacije zahteva neki tip T, osim tamo gde je eksplicitno rečeno drugačije, može se upotrebiti tip const T.
  - svuda gde se kao operand neke operacije zahteva tip T*, osim tamo gde je eksplicitno rečeno drugačije, može se upotrebiti tip T *const.
  - svuda gde se kao operand neke operacije zahteva tip const T, osim tamo gde je eksplicitno rečeno drugačije, može se upotrebiti tip T.

```
const int a=5;
int b=3;
int c;
//...
c=a+b*3;
```

- Pri pozivu funkcije formalni argumenti poziva funkcije inicijalizuju se odgovarajućim stvarnim argumentima.
  - Pokazivač tipa const T* se može inicijalizovati pokazivačem tipa T*, ali obrnuto ne važi.

```
char *s="asdfgh";      // s je pokazivač na nekonstantni objekat,
int i;
                       // neka je formalni argument f-je strlen
                       //      int strlen (const char *s);
                       // je pokazivač na konstantni objekat,
i=strlen(s);           // pa se ovako može pozvati
```

Nizovi znakova i pokazivači na znakove

- Ako je strlen deklarisana kao: `int mystrlen (char *s);`

  onda ne bi mogla da se poziva sa stvarnim argumentom koji je tipa const char*:

```
const char *s="asdfgh"; //s je pokazivač na konstantni objekat,
int i;                  //
i=mystrlen(s);            // greška
i=mystrlen("asdfgh");     // ok
```

- Prevodilac obezbeđuje kontrolu programera "da ispunjava data obećanja".
- Programer je naveo da je s pokazivač na konstantni objekat, što znači da je "obećao" da se taj objekat neće menjati.
  - U funkciji strlen (`int mystrlen (char *s);`) nema "obećanja" da se objekat na koji ukazuje formalni argument neće menjati, pa prevodilac ne garantuje da ta izmena i ne postoji u telu funkcije.
  - Kada bi se ovakav poziv dozvolio, postojala bi mogućnost da se objekat na koji ukazuje formalni argument promeni, a to je isti objekat na koji ukazuje s, pa postoji mogućnost da se početno obećanje prekrši
  - Ako se pokazivač na konstatni objekat inicijalizuje pokazivačem na nekonstatni objekat, nema opasnosti, već postoji samo "obećanje" da se taj objekat u datom delu programa neće menjati, za ostale delove programa to se ne garantuje.
- Eksplicitnom konverzijom može se pokazivač na nekonstatni objekat inicijalizovati pokazivačem na konstatni objekat (odricanje konstatnosti).

Strukture

- Strukture su klase kod kojih su svi članovi podrazumevano javni (*public*).
  - Strukture u jeziku C++ treba shvatiti samo kao specijalni slučaj korisničkih tipova, klasa.
- Deklaracija strukture izgleda potpuno isto kao i deklaracija klase, osim što počinje ključnom rečju struct.
- Struktura može imati podatke članove i funkcije članice, konstruktore i destruktore.
- Osnovna razlika između struktura i klasa je, kao što je rečeno, u tome što su članovi strukture javni, osim ako se eksplicitno ne deklarišu kao privatni (*private*) ili zaštićeni (*protected*); članovi klasa su podrazumevano privatni.
- Ime strukture je, kao i ime klase, ime tipa:

```
struct S {          // deklaracija strukture S
  int is;           // svi članovi strukture su podrazumevano javni
  int js;
  S(int,int);       // konstruktor
};

S::S (int  i, int j) {is=i; js=j;}  // konstruktor

void main(){
  S x(1,1),y(2,1),z(3,1);   // objekti tipa S
  //...
  x.is=4;           // ovo može, jer je is javni član
}
```

Strukture

- Strukture vode svoje poreklo iz jezika C:
  - tamo su one bile korišćene za realizaciju potrebnih struktura podataka.
- U jeziku C++ strukture, kao i klase, mogu da imaju i funkcije članice.
- U jeziku C++ strukture se ređe koriste, uglavnom tamo gde je potrebno realizovati neku jednostavniju strukturu podataka koja služi samo za skladištenje podataka, a nema sve potrebne odlike "prave" klase (nema funkcije članice).
- Ovakve strukture često imaju samo konstruktore kao funkcije članice, koji inicijalizuju njihove podatke članove na potrebne vrednosti.

```
struct Slog {
  char *ime;
  char *lk;
  int  godine;

  Slog (const char*, const char*, int);
};
```

Unije

- Unija je struktura kod koje se svi podaci članovi objekta koji pripada toj uniji kao tipu, smeštaju u memoriju počev od iste adrese, tako da se preklapaju.
- Objekat tipa unije u svakom trenutku sadrži samo jedan od svojih podataka članova.
- Unija zauzima prostor dovoljan da se smesti svaki od njenih podataka članova.
- Deklaracija unije počinje ključnom rečju union. Unija može posedovati svoje konstruktore, destruktore i funkcije članice.

```
union U {
  int i;        // svi ovi članovi počinju od iste adrese
  char c;
};
void main () {
  U u;
  u.i=1;        // od sada postoji u.i
  int j=u.i+1; // ovo je u redu, jer u.i postoji
  u.c='a';      // od sada postoji u.c a ne postoji u.i
  j+=u.i;       // rezultat je nedefinisan, jer u.i nije vise validno!
}
```

- Unija ne može biti izvedena iz neke klase, niti se iz unije može izvesti neka klasa. Unija ne može imati virtuelne funkcije članice, niti može imati član koji je objekat klase koja ima konstruktor, destruktor ili predefinisan operator dodele. Unija ne može imati statičke podatke članove.

Unije

- Posebna vrsta unije je tzv. anonimna unija. To je unija koja ne nosi ime.
- Deklaracijom ovakve unije definiše se jedan bezimeni objekat tipa te unije, a ne tip.
- Imena članova te unije moraju biti različita od ostalih imena u istoj oblasti važenja, a članovima ovakve unije se pristupa direktno, bez uobičajene sintakse za pristup članu (operator .).

```
void f() {
  union
  {      // bezimena unija ne definise tip vec bezimeni objekat
    int i;  // u funkciji f ne mogu postojati druga imena i i c
    char c;
  };
  i=3;          // pristup članu i bezimene unije
  c='a';        // pristup članu c bezimene unije
}
```

- Globalna anonimna unija mora biti deklarisana kao static.
- Anonimna unija ne može imati privatne ni zaštićene članove, niti funkcije članice.
- Unija za koju se definišu objekti ili pokazivači nije anonimna unija

```
union {int i; char c;} u, *pu=&u;  // objekat i pokazivač na uniju
i=1;       // greška, jer unija nije anonimna!
u.c='a';   // ovako može,
pu->i=3;    // ili ovako
```

Unije

- Unije se koriste u sistemskim programima za definisanje struktura podataka niskog nivoa, u cilju uštede prostora u memoriji računara, ili kada neka struktura podataka nikada ne čuva više od jednog člana nekog tipa.
- Objekti tipa unije najčešće upotrebljavaju kao članovi neke druge strukture podataka.

```
struct Osoba {
  char *ime;
  int god;
  char pol;
  union {
    char *devojackoprezime;
    char vojni_rok;
  } podatak;
};
```

  - Osoba ima: ime, godine starosti i pol.
  - Ako član pol ima vrednost 'M', radi se o osobi muškog pola i tada koristimo unijin član vojni_rok.
  - Ako član pol ima vrednost ‘F’ radi se o osobi ženskog pola i tada koristimo unijin član devojackoprezime.
  - Programer se stara o napred navedenim stavkama.

Bit polja

- Posebna vrsta podataka članova su tzv. bit polja.
- Podatak član koji je deklarisan kao bit polje se na određeni način, koji zavisi od implementacije prevodioca, smešta unutar objekta klase. Širina bit polja deklariše se navođenjem konstantog izraza iza dvotačke (:).

```
struct S
{
  int p1 : 12;      // polje p1 širine 12 (bita)
  int p2 : 8;       // polje p2 širine 8 (bita)
  int p3 : 4;       // polje p3 širine 4 (bita)
};

void set(S *ps, int mode)
{
  ps->p1=4;         // pristup članu p1 objekta *ps
  ps->p2=2*mode+1; // pristup članu p2 objekta *ps
}
```

- Način smeštanja bit polja unutar objekata zavisi od prevodioca.
- Bit polja se redom pakuju u adresibilne jedinice memorije računara.
  - Poravnanje bit polja na memorijske reči je takođe zavisno od implementacije prevodioca.
  - Na nekim mašinama, bit polja se smeštaju u memorijske reči sleva u desno, dok na drugim suprotno.

Bit polja

- Bit polje koje nema ime, već samo označenu širinu, služi da forsira odgovarajuće poravnanje smeštanja ostalih bit polja u memoriju računara.
  - Bezimeno polje širine 0 forsira poravnanje sledećeg bit polja na celu adresibilnu jedinicu memorije računara. (`int :0;`)
  - Bezimeno bit polje nije član klase i ne može se inicijalizovati.
- Bit polje mora biti celobrojnog tipa.
  - Od implementacije zavisi da li će se bit polje tipa int tretirati kao označeno ili neoznačeno.
  - Operator uzimanja adrese (&) ne može biti primenjen na bit polje.

- Koriste se za definiciju značenja pojedinih polja bitova unutar memorijskih reči.
- Bit polja se koriste namenski, za datu mašinu, pa su programi koji koriste bit polja najčešće neprenosivi.
- Jezik C++ je zadržao bit polja iz razloga kompatibilnosti, kao i za potrebe realizovanja nižih slojeva sistemskih programa.

Opšti slučaj izvođenja tipova

- U dosadašnjem prikazu izvedenih tipova korišćeni su iskazi poput "ako je T neki tip, onda je T* izvedeni tip: pokazivač na T".
    - Ovakve definicije dozvoljavaju proizvoljno rekurentno izvođenje tipova: izvedeni tipovi u jeziku C++ mogu biti proizvoljno složeni.
- Gramatikom jezika C++ precizno je definisano kako se tipovi izvode i na koji način prevodilac tumači složene tipove.
    - Ova pravila su formirana za automatsko prevođenje, pa nisu pogodna za ljudsku upotrebu u čitanju programa.

Koji je tip ovde u pitanju ?

```
int*(*[])(int);
```

"niz pokazivača na funkciju tipa int* int".

```
int*(*(*)(int))(int);
```

"pokazivač na funkciju koja uzima argument tipa int i vraća rezultat tipa pokazivača na funkciju koja uzima argument tipa int i vraća rezultat tipa pokazivač na int".

- Naravno da su ova dva navedena tipa verovatno besmislena ali su podesni za razumevanje čitanja c++ koda.

Opšti slučaj izvođenja tipova

- Deklaracija objekta x tipa za navedene tipove je:

```
int*(*x[])(int);
int*(*(*x)(int))(int);
```

- U primeru:

```
const int *(*pf)(int), i=5, *pi=&i, *a[10], f(int*);
pf   "pokazivač na funkciju koja uzima argument tipa int
     i vraća tip pokazivač na const int"

i    "const int"

pi   "pokazivač na const int",

a    "niz od 10 elemenata čiji su elementi tipa pokazivač na
      const int",

f    "funkcija koja uzima argument tipa pokazivač na int i
      vraća tip const int".
```

- Objektno orijentisano programiranje nudi pojam klasa kao korisničkog tipa, koji umanjuje potrebu za složenim izvođenjem tipova: sva složenost realizacije nekog tipa se može "lepo upakovati" u klasu.

Primer 1/4

```
//_____________________________fajl gde je main
#include <iostream>
#include <string>
#include "Automobil.h"
using namespace std;
const int UKUPNO_AUTOMOBILA = 3;
void IspisiPodatkeOAutomobilima(Automobil* autokuca,
                                int ukupnoautomobila){
  for(int i=0; i<ukupnoautomobila; ++i){
    cout<<"Automobil: "<<i+1<<endl;
    //(autokuca+i)->IspisiPodatke(); // moze i ovako
    autokuca[i].IspisiPodatke();     // i ovako
    cout<<endl;
  }
}
int main(){
  string modelautomobila;
  double cenaautomobila;
  Automobil autokuca[UKUPNO_AUTOMOBILA];
  IspisiPodatkeOAutomobilima(autokuca,UKUPNO_AUTOMOBILA);
```

Primer 2/4

```
  for(int i=0; i<UKUPNO_AUTOMOBILA; ++i)
  {
    cout<<"Unesite model "<<i+1<<". automobila = ";
    getline(cin,modelautomobila);
    cout<<"Unesite cenu  "<<i+1<<". automobila = ";
    cin>>cenaautomobila; cin.ignore();
    autokuca[i].PostaviPodatke(modelautomobila, cenaautomobila);
  }
  IspisiPodatkeOAutomobilima(autokuca,UKUPNO_AUTOMOBILA);
  double ukupnacena = 0;
  for(int j=0; j<UKUPNO_AUTOMOBILA; ++j)
    ukupnacena+=autokuca[j].UzmiCenu();
  cout<<"Ukupna cena svih automobila je "
      <<ukupnacena<<" EUR"<<endl;
  system("pause");
  return(0);
}
```

Primer 3/4

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

class Automobil
{
 private:
   string model;
   double cena;
 public:
   Automobil();
   Automobil(string, double);
   void PostaviPodatke(string,double);
   void IspisiPodatke() const;
   double UzmiCenu() const;
};
```

Primer 4/4

```
#include "Automobil.h"
Automobil::Automobil()
{
   model = "nema naziv";
   cena = 0;
}
Automobil::Automobil(string m, double c){ model = m;
                                          cena = c;
}
void Automobil::PostaviPodatke(string nazivmodela ,
                               double cenaautomobila){
  model = nazivmodela;
  cena = cenaautomobila;
}
void Automobil::IspisiPodatke() const{
  cout<<model<<", "<<cena<<" EUR"<<endl;
}

double Automobil::UzmiCenu() const{  return cena;  }
```

Izlaz

```
Automobil: 1
nema naziv, 0 EUR

Automobil: 2
nema naziv, 0 EUR

Automobil: 3
nema naziv, 0 EUR

Unesite model 1. automobila = alfa
Unesite cenu  1. automobila = 15000
Unesite model 2. automobila = bmw
Unesite cenu  2. automobila = 20000
Unesite model 3. automobila = audi
Unesite cenu  3. automobila = 22000
```

```
Automobil: 1
alfa, 15000 EUR

Automobil: 2
bmw, 20000 EUR

Automobil: 3
audi, 22000 EUR

Ukupna cena svih automobila je 57000 EUR
Press any key to continue . . .
```

Primer 1/2

```
//samo zbog boljeg uocavanja, sve je u fajlu gde je main
#include <iostream>
using namespace std;
class C
{
  public:
   int a;
   static int b; // radimo detaljno kasnije
   C(){a=0;}
};
int C::b=100; //radimo detaljno kasnije

const int a[2]={10,20};

int mystrlen(char* p)
{
  int i=0;
  for(;*p++;i++);
  return i;
}
```

Primer 2/2

```
int main(){
  const C c;
  //c.a=100;//c.a nije staticni
  c.b=1000; //OK, c.b je staticni
  //a[1]=2000;//ne sme, a[] je konstantan
  const char *s="asdfgh";
  int i;

  //i=mystrlen(s);cout<<"i="<<i<<endl;
  //bilo bi nepostovanje konstantosti

  i=mystrlen("asdfgh");cout<<"i="<<i<<endl;     // ok
  system("pause");
  return 0;
}
```

Primer 1/4

```
//fajl MojeKonstante.h
  #pragma once
  #define PRVA_RECENICA 1
  #define DRUGA_RECENICA 2
  const int desetka=10;

//fajl Vic.h
#pragma once
class Vic{
    private:
      char* prvarecenica;
      char* drugarecenica;
    public:
      Vic();
      void PostaviRecenicu(int rednibrojrecenice,
                           char* recenica);
      void IspisiRecenice();
};
```

Primer 2/4

```
#include <iostream>
using namespace std;
#include "MojeKonstante.h"
#include "Vic.h"

Vic::Vic(){ prvarecenica = drugarecenica = "prazno"; }
void Vic::PostaviRecenicu(int rednibrojrecenice,char* recenica){
  switch(rednibrojrecenice)
  {
    case PRVA_RECENICA : prvarecenica  = recenica; break;
    case DRUGA_RECENICA: drugarecenica = recenica; break;
    default: cout<<"nema "<<rednibrojrecenice<<" recenice!!!";
  }
}

void Vic::IspisiRecenice(){
  cout<<prvarecenica<<endl;
  cout<<drugarecenica<<endl;
  cout<<"ocena je "<<desetka<<endl;
}
```

Primer 3/4

```
#include <iostream>
using namespace std;
#include "MojeKonstante.h"
#include "MojeKonstante.h"          //ok, moze opet
#include "Vic.h"

int main(int argc, char* argv[])
{
  for(int i=0; i<argc; ++i)
  {
    cout<<"argv["<<i<<"]="<<argv[i]<<endl;
  }
  cout<<"Konstanta desetka="<<desetka<<endl;

  Vic v1;
  v1.IspisiRecenice();cout<<endl;
  v1.PostaviRecenicu(PRVA_RECENICA,"Govori ko ti je saucesnik!");
```

Primer 4/4

```
  v1.PostaviRecenicu(DRUGA_RECENICA,"Nisam blesav da izdam
rodjenog brata!");

  Vic v2;
  v2.PostaviRecenicu(PRVA_RECENICA,"Chuck Norris ne jede med.");
  v2.PostaviRecenicu(DRUGA_RECENICA,"On zvace pcele.");

  Vic *pvic;
  pvic = &v1;
  pvic->IspisiRecenice();cout<<endl;
  pvic = &v2;
  pvic->IspisiRecenice();cout<<endl;
  pvic->PostaviRecenicu(PRVA_RECENICA,"OOP");
  pvic->PostaviRecenicu(DRUGA_RECENICA,"Lako je.");
  pvic->IspisiRecenice();
  system("pause");
  return 0;
}
```

Izlaz

```
argv[0]=C:\__Projects\CPP\Proba\Debug\Proba.exe
Konstanta desetka=10
prazno
prazno
ocena je 10

Govori ko ti je saucesnik!
Nisam blesav da izdam rodjenog brata!
ocena je 10

Chuck Norris ne jede med.
On zvace pcele.
ocena je 10

OOP
Lako je.
ocena je 10
Press any key to continue . . .
```