11 - ...Parametarski polimorfizam

Saša Malkov

Odlomak iz knjige Razvoj softvera

(u pripremi)

11.1 Polimorfizam

Za neki *izraz* ili *vrednost* u programu kažemo da su *polimorfni* ako mogu da imaju za domen više različitih tipova. U skladu sa time, za neki *programski kod* kažemo da je *polimorfan* ako sadrži polimorfne vrednosti ili izraze.

Kod slabo tipiziranih programskih jezika, provera tipova, ili bar njen veći deo, se ostavlja za fazu izvršavanja programa. Programi se pokrenu i onda se u toku rada proverava da li svaki pojedinačan sadržaj ima onaj tip koji je neophodan da bi izvršavanje moglo da se nastavi. Isto važi i za polimorfne programe i delove programa.

Sa strogo tipiziranim programskim jezicima stvari stoje drugačije. Kod njih sve provere tipova moraju da se izvrše unapred, u fazi prevođenja (ako se radi o klasičnim prevodiocima) ili u fazi pripreme (ako se radi o interpretatorima). U trenutku izvršavanja programa svaka konkretna vrednost mora da ima tačno jedan konkretan tip i svaki konkretan izraz izračunava vrednost tačno jednog konkretnog tipa. Naravno, i dalje u zavisnosti od konteksta u kome se posmatrani deo koda koristi, isti polimorfni izrazi u različitim slučajevima mogu imati različite konkretne tipove.

Razmotrimo, na primer, sledeći jednostavan polimorfan isečak programskog koda:

```
a = b + c;
```

Kog tipa su promenljive `a`, `b` i `c`? Kog tipa je izraz `b+c`? Ako posmatramo samo navedeni isečak koda, ne možemo da precizno procenimo navedene tipove. Mogli bismo da zaključimo da `a`, `b` i `c` moraju da budu istog tipa, kao i da za taj tip postoje definisani operatori sabiranja i dodeljivanja, ali takvo zaključivanje ne bi bilo dovoljno uopšteno. Tačnijim zaključivanjem bismo mogli da izvedemo sledeći uopšteniji skup uslova koji moraju da budu ispunjeni:

- postoji neki binarni operator „+“, koji za prvi argument tipa *T1* i drugi argument tipa *T2* izračunava rezultat tipa *T3*;
- ime `b` je tipa *Tb*, za koji važi da je podtip od *T1*[30] ili postoji implicitna konverzija iz tipa *Tb* u *T1*;
- ime `c` je tipa *Tc*, za koji važi da je podtip od *T2* ili postoji implicitna konverzija iz tipa *Tc* u *T2*;
- postoji operator dodeljivanja koji promenljivoj tipa *TV* dodeljuje vrednost tipa *TE*;
- *T3* je podtip od *TE* ili postoji implicitna konverzija iz tipa *T3* u *TE*;
- ime `a` je tipa *Ta*, koji je podtip od *TV*[31].

Prethodni primer može da nam posluži da donekle sagledamo složenost postupka proveravanja tipova. Vidimo da je čak i za sasvim jednostavne izraze to relativno složen postupak, u kome figuriše mnogo uslova koji moraju da se usklade. skup uslova bi bio još mnogo složeniji kada bismo formalno definisali pojam *podtipa* i uveli u razmatranje i sva odgovarajuća pravila koja bi iz takve definicije sledila. Saznanje o ovoj složenosti može da nam pomogne da imamo malo više razumevanja

[30] Primetimo da je specijalan slučaj podtipa upravo jednakost tipova. Ako je *A* podtip tipa *B* i *B* podtip tipa *A*, onda su *A* i *B* jednaki tipovi.

[31] Pri razmatranju tipa operatora dodeljivanja moramo da razlikujemo formalnu ispravnost od faktičke ispravnosti. Pri proveravanju tipova može da se razmatra samo formalna ispravnost. Operator dodeljivanja očekuje da dobije kao argument vrednost tipa *T3*, koja može da se menja – u slučaju C++-a to je referenca na *T3*. Tip *Ta* će biti u skladu sa takvim zahtevom ako je podtip tipa *TV*. Sa druge strane, znamo da dodeljivanje neće biti ispravno izvršeno ako je *Ta* pravi podtip, tj. nije upravo *TV*. Ali to nije na proveri tipova, već na programeru, da obezbedi da postoji i odgovarajući operator dodeljivanja. Provera tipova proverava ispravnost, a semantika programskog jezika određuje koja od *ispravno primenljivih* verzija nekog operatora ili funkcije će zaista da se primeni.

prema određenoj sporosti koju iskazuju prevodioci kada se suoče sa polimorfnim izrazima.

Polimorfizam ima veliki značaj za programiranje. Ako programiranje posmatramo kao postupak formalnog modeliranja nekih procesa, izraza ili odnosa u posmatranom domenu, onda polimorfizam predstavlja jedno od osnovnih tehničkih sredstava za apstrahovanje predmeta modeliranja. Zbog takve uloge, polimorfizam u svakodnevnoj programerskoj praksi predstavlja veoma korisno sredstvo za ostvarivanje višestruke upotrebljivosti koda. Pisanjem polimorfnog koda dobija se programski kod koji može da se upotrebi u različitim kontekstima, za različite tipove podataka, pa čak i za veoma raziličite postupke.

Sveprisustnost objektno orijentisanog programiranja i objektno orijentisanih metodologija ima za posledicu da se čak i u stručnoj literaturi polimorfizam često poistovećuje sa konceptom nasleđivanja, što je daleko od istine. Iako je hijerarhijski polimorfizam čvrsto ugrađen u samu suštinu objektno orijentisanog programiranja kroz koncept nasleđivanja klasa, to je ipak samo jedan od vidova polimorfizma. U savremenom razvoju softvera se prepoznaju četiri osnovne vrste polimorfizma:

- hijerarhijski polimorfizam;
- parametarski polimorfizam;
- implicitni polimorfizam i
- ad-hok polimorfizam.

Ukratko ćemo predstaviti različite vrste polimorfizma, a zatim ćemo se više posvetiti parametarskom polimorfizmu.

Hijerarhijski polimorfizam

Objektno orijentisano programiranje počiva na konceptima klase i nasleđivanja klasa. Nasleđivanje klasa u osnovi nije ništa drugo do deklarisanje da jedna klasa predstavlja specijalan slučaj (ili *potklasu*) druge klase, tj. da druga klasa predstavlja uopštenje (tj. *natklasu*) prve klase. Relacija *potklasa* je specijalan slučaj relacije *podtip*, a relacija *natklasa* specijalan slučaj relacije *nadtip*. Ako tipove posmatramo kao skupove, onda je skup svih vrednosti nekog *podtipa* podskup skupa vrednosti njegovog *nadtipa*. Zato se nasleđivanje često opisuje kao odnos *jeste* (engl. *is a*), zato što neki objekat *potklase* jeste i objekat *natklase*.

Hijerarhijski polimorfizam koristi formalno (tekstom programskog koda, a u skladu sa sintaksom programskog jezika) uspostavljene hijerarhijske odnose *nadtip* i *podtip* među klasama kao sredstvo ostvarivanja polimorfizma. On je u osnovi osobine objektno orijentisanih programskih jezika da programski kod, koji je napisan da radi sa objektima jedne klase, može da radi i sa objektima svih njenih potklasa.

Na primer, ako u baznoj klasi hijerarhije geometrijskih likova postoji metod `Polozaj()` koji izračunava položaj lika, onda će on moći da se upotrebi i na objektima bilo koje konkretne klase geometrijskih likova, koja je neposredno ili posredno izvedena iz te bazne klase:

```
void f( Kvadrat& k ){
    ... k.Polozaj() ...
}
```

Kao što smo već istakli i predstavili primerom (u odeljku 3.4 – *Slabosti objektno orijentisanih koncepata*, na strani 32), odnosi koji upravljaju nasleđivanjem nisu do kraja formalno definisani, što ima neke veoma neugodne posledice. Činjenica da u izvedenim klasama neki metodi mogu da se napišu tako da ne rade isto kao u baznoj klasi zapravo narušava osnovne pretpostavke hijerarhijskog polimorfizma (relaciju *jeste*).

Parametarski polimorfizam

Druga značajna vrsta polimorfizma je parametarski polimorfizam. On nema praktično nikakve veze sa hijerarhijskim polimorfizmom. Za razliku od hijerarhijskog polimorfizma, on ne počiva na deklarisanim odnosima među tipovima ili klasama. Parametarski polimorfizam počiva na upotrebi tzv. *tipskih promenljivih*. Tipske promenljive se koriste za simboličko označavanje tipova vrednosti, promenljivih i izraza. Svaki put kada se upotrebi kod, koji je napisan uz upotrebu parametarskog polimorfizma, simbolički tipovi (tipske promenljive) se zamenjuju konkretnim tipovima i program se prevodi u nepolimorfnom obliku.

Da bi prevođenje bilo uspešno, neophodno je da u svim delovima tog polimorfnog koda zamenjivanje simboličkih tipova konkretnim tipovima proizvodi ispravan programski kod. Ako to nije slučaj, tj. ako je za konkretan tip izabran neki tip za koji dati programski kod nije ispravan, onda će prevodilac prijaviti grešku. U zavisnosti od programskog jezika (ili čak verzije programskog jezika), ispravnost se ustanovljava ili pokušavanjem prevođenja i ustanovljavanjem da li je ono uspešno izvedeno ili ne, ili proveravanjem nekih eksplicitno deklarisanih uslova, ili oboje.

Određivanje konkretnih tipova, koji se koriste u nekom konkretnom slučaju, može da se izvodi eksplicitno, navođenjem tipa, ili implicitno, na osnovu pravila prevođenja konkretnog programskog jezika.

Parametarski polimorfizam se često naziva i *generičkim programiranjem*. Kada se govori o tehnici programiranja, onda je termin generičko programiranje daleko češće u upotrebi, a kada se govori o tehnikama ostvarivanja polimorfizma i višestruke upotrebljivosti programskog koda, onda se obično koristi termin parametarski polimorfizam. U svakom slučaju, radi se o suštinski istim stvarima.

Parametarski polimorfizam je zastupljen u sve većem broju programskih jezika. Smatra se da se po prvi put pojavio 1976. godine u okviru funkcionalnog programskog jezika *ML*. Predstavljao je važnu tehniku programiranja i u programskom jeziku *Ada*. Tokom 1980-ih je uveden u veliki broj funkcionalnih jezika, pa možemo reći da danas većina značajnih funkcionalnih programskih jezika koristi neki oblik parametarskog polimorfizma, sa izuzetkom onih jezika koji umesto parametarskog koriste implicitni polimorfizam.

Programski jezik C++ je od samog početka podržavao parametarski polimorfizam, ali njegov značaj se vremenom povećavao. Ključan doprinos je ostvarila grupa istraživača koji su radili na oblikovanju i razvoju biblioteka. Među njima se izdvajaju Aleksandar Stepanov i Meng Li, koji su početkom 1990-ih definisali *Standardnu biblioteku šablona* (engl. *Standard Template Library*), koja se (u nekom od svojih oblika) distribuirala uz većinu savremenih prevodilaca [Step1994]. Biblioteka je potom došla i do tela za standardizaciju programskog jezika C++ i od nje je nastala Standardna biblioteka programskog jezika C++, što je formalizovano ISO standardom C++98. Standardna biblioteka C++-a skoro u potpunosti počiva na intenzivnoj upotrebi parametarskog polimorfizma. Po uzoru na C++, parametarski polimorfizam su, na sličan način ali uz značajna ograničenja, podržali i programski jezici *Java* i *C#*.

Podrška za parametarski polimorfizam u programskom jeziku *C++* je implementirana kroz koncept *šablona funkcija* i *klasa*. *Šablon funkcije* predstavlja funkciju u kojoj se upotrebljava neki parametarski (simbolički) tip. Slično tome, *šablon klase* je definicija klase koja počiva na upotrebi jednog ili više simboličkih tipova. Ove dve vrste šablona predstavljaju veoma važnu tehniku programiranja, pa ćemo ih u ovom poglavlju malo temeljnije obraditi.

Implicitni polimorfizam

Implicitni polimorfizam podrazumeva da se pri pisanju koda uopšte ne navode tipovi vrednosti i izraza. Pretpostavlja se da će prevodilac analizirati svaki konkretan segment programskog koda i sam zaključiti za koje tipove može da se prevede. Ako neki deo koda može da se uspešno prevede za više različitih tipova, onda se prevodi i koristi kao polimorfan deo koda.

Implicitni polimorfizam predstavlja uopštenje parametarskog polimorfizma. Implicitni polimorfizam se koristi u velikom broju dinamičkih programskih jezika, kod kojih se proveravanje tipova izraza (a time i proveravanje ispravnosti koda) odlaže do faze izvršavanja programa (na primer *Lisp*), ali i kod značajnog broja programskih jezika kod kojih se statičkom proverom tipova u fazi prevođenja tačno ustanovljava (potencijalno polimorfan) tip svakog izraza u kodu.

Na primer, naredna definicija funkcije `max` u programskom jeziku *Wafl* je polimorfna radi za sve tipove za koje postoji definisan binarni operator „`>`“:

```
max(x,y) = if x>y then x else y;
```

Još opštija bi bila definicija funkcije koja pronalazi najmanji element liste elemenata, pri čemu se za poređenje koristi binarna funkcija `less`, a u slučaju prazne liste se vraća `zero`:

```
min(lst,less,zero) =
    foldl(
        lst,
        \cmin,x [less]: if less(cmin,x) then cmin else x,
        zero
    );
```

Zanimljivo je da primetimo da specifičan vid implicitnog polimorfizma predstavljaju i makroi programskog jezika C. Na primer, ako definišemo makro `max` na uobičajen način:

```
#define max(x,y) ((x)>(y) ? (x) : (y))
```

onda napisani makro možemo da upotrebimo za izračunavanje većeg od dva argumenta za bilo koje tipove podataka za koje je definisan operator poređenja „`>`“.

Ad-hok polimorfizam

Ad-hok polimorfizam počiva na osobini nekih programskih jezika da dopuštaju višeznačnost imena funkcija i operatora (engl. *overloading*). Ako se napiše više funkcija, tako da imaju isto ime i isti broj argumenata, ali su im argumenti različitih tipova, pri čemu sve funkcije rade suštinski istu stvar, prilagođenu različitim tipovima argumenata, onda one predstavljaju primer ad-hok polimorfizma.

U većini programskih jezika višeznačnost imena funkcija i operatora je ograničena samo na ugrađene osnovne elemente jezika (npr. operator sabiranja), ali postoje programski jezici koji programerima dopuštaju da sami pišu višeznačne operatore i funkcije. U programskom jeziku C++ dopušteno je da se napiše više funkcija sa istim imenom, sve dok se sve takve funkcije mogu nedvosmisleno razlikovati po broju ili tipovima argumenata. Na primer, možemo da napišemo dve različite implementacije funkcije `potencijalnoZanimljivBroj`:

```
int potencijalnoZanimljivBroj( int n ){
    return n%3==0 && n%5!=0;
}
double potencijalnoZanimljivBroj( double x ){
    return potencijalnoZanimljivBroj( round(x) );
}
```

Ad-hok polimorfizam deli neke osobine parametarskog polimorfizma, pa ga neki autori smatraju njegovim posebnim slučajem i ne prepoznaju kao posebnu vrstu

polimorfizma. Funkcije koje dele zajedničko ime, moraju biti *konceptualno usklađene* da bi se moglo govoriti o vidu polimorfizma. Ako neke funkcije imaju isto ime, ali za različite tipove argumenata izračunavaju suštinski različite stvari ili izvode suštinski različite operacije, onda nije u pitanju polimorfizam, nego samo višeznačnost imena funkcije. Činjenica da programer mora da se stara o toj konceptualnoj usklađenosti predstavlja značajan faktor razlikovanja između ad-hok i parametarskog polimorfizma.

Primena ad-hok polimorfizma može da se lepo nadopunjuje sa parametarskim polimorfizmom. Na primer, ako se napiše generička funkcija `g` (primena parametarskog polimorfizma) koja koristi neku funkciju `f` za parametarski tip `T`, onda se eksplicitnim definisanjem funkcije `f` za neke tipove `T1` i `T2` (primena ad-hok polimorfizma) omogućava da se i funkcija `g` primenjuje na `T1` i `T2`. Drugi vid je tzv. specijalizacija šablona, o kojoj će biti više reči u daljem tekstu.

11.2 Šabloni funkcija

Šabloni fukcija u programskom jeziku C++ predstavljaju sredstvo za pisanje polimorfnih funkcija. Polimorfizam se ostvaruje apstrahovanjem nekih tipova i/ili konstanti koji se pojavljuju u deklaraciji i implementaciji funkcije. Najčešće se apstrahuju tipovi argumenata i rezultata, ili konstante koje se odnose na veličinu ili strukturu argumenata ili rezultata, ali se mogu apstrahovati i tipovi i konstante koji se koriste u samoj implementaciji.

Deklaracijama i definicijama šablona funkcija mora da prethodi *deklaracija parametara šablona*. Deklaracija parametara šablona se sastoji od ključne reči `template`, iza koje se u uglastim zagradama navode parametri šablona. Svaki parametar se opisuje *vrstom* i *imenom*. Parametar šablona može da predstavlja tip (tzv. *tipski parametar*) ili konstantu (tzv. *konstantni parametar*). Ako se radi o tipu, onda se kao oznaka vrste upotrebljava ključna reč `typename`[32], dok se u slučaju konstante kao oznaka vrste upotrebljava *tip konstante*. Na primer, u sledećoj deklaraciji šablona se navodi da postoje dva parametra – tip `T` i celobrojna konstanta `K`:

[32] Umesto ključne reči `typename` može da se upotrebi i ključna reč `class`. U prvim verzijama programskog jezika C++ je u ovom kontekstu upotrebljavana samo ključna reč `class`, prvenstveno da se ne bi uvodile nove ključne reči. Ipak, kada je međunarodni komitet radio na standardizaciji, procenili su da je ovakva upotreba ključne reči `class` problematična, najpre zbog toga što parametrizovani tipovi ne moraju biti samo klase već mogu da budu i ugrađeni tipovi, funkcije i drugo, ali i zato što ova primena ključne reči nema mnogo veze sa njenim osnovnim značenjem. Zbog toga je uvedena nova ključna reč `typename` i preporučena je njena upotreba u ovom kontekstu, iako je zbog kompatibilnosti sa starim izvornim kodim omogućeno da se i dalje upotrebljava i ključna reč `class`.

```
template< typename T, int K >
```

Nakon deklaracije šablona navodi se deklaracija ili definicija funkcije. Ni deklaracija ni definicija se ne razlikuju značajno od uobičajenog načina deklarisanja i definisanja funkcija. Jedina razlika je u tome što se i u deklaraciji i u definiciji mogu upotrebljavati parametrizovani tipovi i konstante.

Zamenjivanje parametara šablona konkretnim tipovima i konstantama naziva se *instanciranje šablona*. Rezultat instanciranja šablona je jedna konkretna funkcija, koja može da se prevede i upotrebi[33]. Šabloni funkcije mogu da se instanciraju *eksplicitnim vezivanjem parametara* ili *implicitnim prepoznavanjem parametara*. Eksplicitno vezivanje parametara podrazumeva da programer na mestu upotrebe šablona funkcije eksplicitno navede vrednosti parametara. Vrednosti parametara se navode neposredno iza imena funkcije, u uglastim zagradama, u istom poretku u kome su deklarisani. Na primer, u narednom izrazu se instancira šablon funkcije `f` navođenjem vrednosti parametara `int` i `10`:

```
...f<int,10>(...)...
```

Ako pri pozivanju funkcije, na osnovu navedenih konkretnih argumenata, može da se jednoznačno ustanovi koje su to vrednosti parametara šablona koje daju odgovarajuću instancu šablona funkcije, onda može da se primeni implicitno instanciranje. U tom slučaju šablon funkcije možemo da koristimo kao da je u pitanju obična funkcija, bez eksplicitnog navođenja vrednosti parametara šablona, ili uz eksplicitno označavanje da se radi o šablonu, ali bez navođenja vrednosti parametara:

```
...f<>(...)...
```

Ako programer upotrebi šablon na implicitan način (bez eksplicitnog instanciranja, očekujući da će prevodilac uspeti da jednoznačno prepozna odgovarajuće vrednosti parametara), a prevodilac ustanovi da ne postoji jednoznačno rešenje (t.j. da ne postoji nijedna odgovarajuća konfiguracija parametara, ili da, nasuprot tome, postoji više podjednako prihvatljivih konfiguracija parametara), onda će prevodilac prijaviti odgovarajuću grešku. Takođe, ako eksplicitno navedeni parametri daju definiciju funkcije koja ne može da se prevede (npr. za navedene tipove ne postoje neke od operacija koje se upotrebljavaju u definiciji šablona funkcije), biće izdata odgovarajuća greška pri prevođenju.

[33] Naravno, pod uslovom da definicija funkcije može da se uspešno prevede i izvrši za konkretne vrednosti parametara. Načinu prevođenja šablona i neophodnim uslovima za uspešno prevođenje šablona ćemo se više posvetiti u jednom od narednih odelajka.

Primetimo da obe vrste grešaka pri instanciranju šablona mogu da budu veoma prikrivene i posredno iskazane, posebno u složenim slučajevima, kada se u definiciji jednog šablona upotrebljavaju neki drugi šabloni.

Ako se navede upotreba bez navođenja parametara šablona, pa i bez navođenja uglastih zagrada (tj. u obliku u kome može da se pozove obična funkcija), a pri tome postoji funkcija sa istim imenom i brojem argumenata i saglasnim tipovima argumenata, onda će prevodilac uvek pre birati tu funkciju nego odgovarajuću instancu šablona.

Ne postoje nikakva posebna ograničenja u pogledu broja različitih instanciranja šablona, sve dok su sva pojedinačna instanciranja ispravna. Jedan isti šablon funkcije može da se instancira na različite načine u jednom istom izrazu, bilo eksplicitno bilo implicitno.

Proces prevođenja šablona funkcija ima nekoliko specifičnosti. U „prvom prolazu“, pri „prevođenju“ same definicije šablona funkcije, prevodilac proverava samo da li definicija zadovoljava osnovne normative koje propisuje sintaksa programskog jezika C++. Zbog toga što još uvek nisu poznate vrednosti parametara, nije moguće ni tačno prevođenje ni dosledno proveravanje ispravnosti sintakse. Tek pri instanciranju šablona funkcije prevodilac „ponavlja“ prevođenje šablona za konkretne navedene vrednosti parametara.

Posledica takvog načina prevođenja je da se mnoge greške načinjene pri pisanju definicije šablona funkcije ispoljavaju tek pri njenom instanciranju, što može da otežа prepoznavanje grešaka. Zbog toga je pri testiranju šablona funkcije i posebno pri pisanju testova jedinica koda kojima se proverava ispravnost šablona funkcije, neophodno da se testiranje sprovede na različitim instancama šablona. Štaviše, često je potrebno da se napravi veći broj suštinski različitih instanci.

Šablon funkcije max2

Za ilustraciju ćemo se poslužiti jednostavnom funkcijom `max2`[34], koja računa veći od dva data cela broja:

```
int max2( int x, int y ){
    return x > y ? x : y;
}
```

Funkciju `max2` možemo da zamenimo polimorfnom implementacijom, šablonom funkcije `max2`, na sledeći način:

```
template< typename T >
```

[34] U primeru je upotrebljeno ime `max2` da pri eventualnom prevođenju primera ne bi dolazilo do mešanja sa šablonom funkcije `max` standardne biblioteke.

```
T max2( T x, T y ){
    return x > y ? x : y;
}
```

Primetimo da su sve načinjene izmene sasvim jednostavne. Najpre je ispred same definicije funkcije dopisana deklaracija parametara šablona: `template<typename T>`, a zatim su u samoj definiciji funkcije sva pojavljivanja tipa `int` zamenjena tipskim parametrom `T`. Tako smo dobili polimorfnu implementaciju `max2`.

Šablon funkcije `max2` možemo da instanciramo i upotrebimo uz eksplicitno ili implicitno vezivanje odgovarajućeg tipa kao vrednosti parametra `T`. Ako se tipski parametar `T` zameni tipom `int`, dobija se celobrojna funkcija, koja je po svemu (a pre svega po ponašanju i efikasnosti) identična funkciji od koje smo započeli pravljenje šablona. Ako bismo, umesto toga, tip `T` zamenili nekim drugim tipom, na primer `double`, onda bismo dobili implementaciju koja je odgovarajuća za taj drugi tip. Na primer, eksplicitno instanciranje tipom `int` može da se zapiše:

```
... max2<int>(a,b) ...
```

U narednom primeru se u istom izrazu šablon funkcije `max2` dva puta implicitno instancira na različite načine – najpre kao celobrojna funkcija, sa parametrom `T=int`, a zatim kao realna funkcija, sa parametrom `T=double`:

```
... max2(2,5) + max2(1.3,2.4) ...
```

Naglasili smo da će u slučaju neispravnog implicitnog instanciranja prevodilac prijaviti da je prepoznao grešku. Na primer, ako pokušamo da primenimo implicitno instanciranje šablona funkcije `max2` sa argumentima različitih tipova, poput `max2(1,2.5)`, prevodilac će izdati ovakvu ili sličnu poruku:

```
primer.cpp: In function 'int main()':
primer.cpp:14:21: error: no matching function for call to 'max2(int,
double)'
  cout << max2(1, 2.5) << endl;
                     ^
primer.cpp:14:21: note: candidate is:
primer.cpp:5:17: note: template<class T> const T& max2(const T&,
const T&)
 inline const T& max2( const T& x, const T& y )
                 ^
primer.cpp:5:17: note:   template argument deduction/substitution
failed:
primer.cpp:14:21: note:   deduced conflicting types for parameter
'const T' ('int' and 'double')
  cout << max2(1, 2.5) << endl;
                     ^
```

Ako eksplicitno navedemo tip parametra, bilo `int` ili `double`, prevođenje će uspeti i prva instanca će biti celobrojna funkcija, koja izračunava vrednost `2`, a druga instanca će biti realna funkcija, koja izračunava vrednost `2.5`:

```
...max2<int>(1,2.5)...
...max2<double>(1,2.5)...
```

Radi kompletnosti, naglasićemo da se pri pisanju ovakvih šablona obično radi sa što opštijim tipovima, da bi se prevodiocu ostavio prostor za širu primenu i automatske optimizacije. Zbog toga bi šablon `max2` trebalo pisati na sledeći način, kako bi se izbeglo eventualno kopiranje u slučaju primene na složenije tipove:

```
template< typename T >
const T& max2( const T& x, const T& y ){
    return x > y ? x : y;
}
```

Jednostavni šabloni funkcija se zbog efikasnosti često pišu kao *inline* funkcije, koje se ne prevode kao posebni delovi koda nego se fizički ugrađuju u kod na svakom mestu upotrebe (poput makroa programskog jezika C):

```
template< typename T >
inline const T& max2( const T& x, const T& y ){
    return x > y ? x : y;
}
```

11.3 Šabloni klasa

Kao što šablonima funkcija mogu da se uopšte funkcije, tako i klase mogu da se uopšte pomoću šablona klasa. Šabloni klasa u programskom jeziku C++ predstavljaju sredstvo za pisanje polimorfnih tipova podataka. Polimorfizam se ostvaruje apstrahovanjem nekih tipova i/ili konstanti u deklaraciji i definiciji klase. Najčešće se apstrahuju oni tipovi podataka i konstante koji se odnose na strukturu elemenata klase ili na argumente najvažnijih metoda klase.

Sintaksa je veoma slična sintaksi šablona funkcije: pri opisivanju šablona klase se pre deklaracije i definicije klase navodi deklaracija parametara šablona. Takođe, ako se neki metod klase implementira van definicije klase, ispred implementacije metoda mora da se navede ista deklaracija parametara šablona.

Za razliku od šablona funkcija, u slučaju šablona klasa dopušteno je samo eksplicitno instanciranje. Implicitno instanciranje u opštem slučaju nije moguće, zato što se tačne vrednosti parametara veoma retko mogu implicitno ustanoviti pri deklarisanju ili pravljenju objekata nekog tipa, zato što parametri šablona mogu da utiču na mnoga različita mesta na kojima se objekti i metodi klase upotrebljavaju.

Kao i u slučaju šablona funkcija, ne postoje ograničenja u pogledu broja različitih instanciranja šablona klasa, sve dok su sva pojedinačna instanciranja ispravna.

Proces prevođenja šablona klasa je sličan prevođenju šablona funkcija, ali uz neke dodatne specifičnosti. U „prvom prolazu“, pri „prevođenju“ same definicije šablona klase, prevodilac proverava samo da li definicije klase i svih metoda zadovoljavaju osnovne normative koje propisuje sintaksa programskog jezika. Pri instanciranju šablona klase prevodilac „ponavlja“ prevođenje strukture klase i uočava eventualne probleme u samoj strukturi. Međutim, prevođenje instanci metoda se odlaže i dalje, sve dok se ne naiđe na njihovu konkretnu upotrebu. To u praksi ima dve veoma značajne posledice. Prva je da se prevođenje instance šablona klase praktično distribuira kroz različite module programa, tako da se svaki metod prevodi tek na mestu upotrebe, pa se i greške ispoljavaju pri prevođenju različitih jedinica koda. Druga je da metodi koji se u programu ne koriste (ili se bar ne koriste za konkretnu instancu šablona), neće ni biti prevođeni. Zbog takvog načina prevođenja može da se dogodi da neke greške u definiciji šablona klase ostanu prilično dugo prikrivene, pa čak i da šablon uđe u široku upotrebu, a da neki problem nije uočen.

Ako smo ranije nagovestili da opširno izveštavanje o greškama pri instanciranju šablona funkcija može da bude neugodno, onda na ovom mestu to moramo još više da istaknemo. Zaista, poruke prevodilaca u vezi sa neispravnim instanciranjem šablona klasa mogu da budu mnogo opširnije, a pronalaženje stvarnih uzroka grešaka mnogo teže, nego što je to slučaj sa šablonima funkcija. Uzrok je, naravno, u potencijalno mnogo većoj složenosti šablona klase u odnosu na šablon funkcije, kao i u brojnim međuzavisnostima različitih elemenata šablona klase. Svaki metod šablona klase je u osnovi jedan šablon funkcije, pri čemu većina metoda koristi neke druge metode istog šablona klase, tako da se povećava i dubina na kojoj se problemi mogu ispoljiti.

Prevođenje šablona i ustanovljavanje neispravnog instanciranja (tj. pokušaja instanciranja sa vrednostima parametara koje zapravo nisu dopuštene) se značajno olakšava uvođenjem *koncepata* u standardu C++2020.

Šablon klase Tacka

Šablone klasa ćemo da ilustrujemo jednostavnim primerom klase `Tacka`, koja predstavlja model tačke u trodimenzionalnom prostoru. Pretpostavimo da imamo klasu čije su koordinate celobrojne i koja ima samo konstruktor i odgovarajući operator za ispisivanje:

```
class Tacka
{
public:
    int x,y,z;
    Tacka(int x0, int y0, int z0)
        : x(x0), y(y0), z(z0)
```

```
        {}
};

ostream& operator<<( ostream& ostr, const Tacka& t )
{
    ostr << "(" << t.x << "," << t.y << "," << t.z << ")";
    return ostr;
}
```

Polazeći od ove klase možemo da napravimo šablon klase `Tacka` tako što ćemo da dodamo deklaraciju parametara šablona i da zamenimo odgovarajuće tipove:

```
template <class T>
class Tacka
{
public:
    T x,y,z;
    Tacka(T x0, T y0, T z0)
        : x(x0), y(y0), z(z0)
        {}
};

template <class T>
ostream& operator<<( ostream& ostr, const Tacka<T>& t )
{
    ostr << "(" << t.x << "," << t.y << "," << t.z << ")";
    return ostr;
}
```

Kao što je već istaknuto, šablon klase mora da se instancira eksplicitno, pa ćemo svaki put pri navođenju tipa instance šablona morati da eksplicirno navedemo vrednosti parametara. Na primer, u sledećem isečku koda koriste se celobrojna i realna instanca šablona klase `Tacka`:

```
Tacka<int> t(1,2,3);
cout << t << endl;
cout << Tacka<int>(1.2, 1.3, 1.4) << endl;
cout << Tacka<float>(1.2, 1.3, 1.4) << endl;
```

11.4 Eksplicitna specijalizacija

Šabloni funkcija i klasa, kao što smo već naglasili, mogu da se instanciraju za sve vrednosti parametara za koje definicija može da se prevede. Međutim, nisu retki slučajevi da neki kod može da se prevede ali da rezultat njegovog izvršavanja, odnosno smisao tog koda u kontekstu konkretnih vrednosti parametara, nije odgovarajući.

Vratimo se na ranije definisan šablon funkcije `max2` i razmotrimo šta će biti rezultat njegove primene na pokazivače:

```
int a(2),b(3);
cout << *max2(&a,&b) << endl;
```

U većini slučajeva rezultat će biti broj 2. Razlog je u tome što će rezultat primene šablona `max2` na pokazivače biti pokazivač na podatak koji se nalazi *na većoj adresi* u memoriji. U ovom slučaju tipski parametar `T` je dobio vrednost „`int*`“ i operacija poređenja je primenjena na pokazivače, a ne na objekte na koje oni pokazuju. Većina prevodilaca smešta lokalne podatke na stek tako da prva definisana promenljiva ide na više adrese, pa je tako i u našem slučaju promenljiva `a` smeštena na višoj memorijskoj adresi u odnosu na `b` i predstavlja rezultat funkcije `max2`. Vrednosti promenljivih pri tome ne igraju nikakvu ulogu.

Ako bismo želeli da uporedimo objekte (u ovom slučaju brojeve) i vratimo adresu onog objekta čija je vrednost veća, onda naš šablon funkcije ne bi dobro radio. Jedno moguće rešenje je da napišemo novi šablon, na primer `max_p`, koji bi radio samo sa pokazivačima. Drugo rešenje je da ostavimo prethodnu i dodamo još jednu verziju postojećeg šablona, koja je prilagođena za slučaj sa pokazivačima:

```
template< typename T >
T max2( T x, T y ){
    return x > y ? x : y;
}

template< typename T >
T* max2( T* x, T* y ){
    return *x > *y ? x : y;
}
```

Pravila prevođenja propisuju da prevodilac, ako ima na raspolaganju više verzija nekog šablona koje mogu da se primene, uvek mora da izabere onu čija deklaracija6 (funkcije ili klase) je manje opšta. Alternativno tumačenje je da se bira ona za koju će vrednosti tipskih parametara biti *jednostavniji* tipovi. Ako prevodilac to ne može jednoznačno da prepozna, prijaviće grešku.

U konkretnom slučaju (prethodni primer) prevodilac ima na raspolaganju dve definicije šablona. Prva je opštija, zato što deklaracija funkcije prihvata sve tipove, dok druga prihvata samo pokazivače. Zbog toga će prevodilac, ako može da bira između ove dve implementacije, uvek birati drugu. Po alternativnom tumačenju, ako bi prevodilac primenio prvu definiciju, tipski parametar `T` bi imao vrednost `int*`, a ako bi primenio drugu, tipski parametar `T` bi imao vrednost `int`. Zato što je tip `int` *jednostavniji* od tipa `int*`, prevodilac bi birao drugu verziju. Nećemo sada ulaziti u formalno definisanje poređenja tipova i složenosti tipova, već ćemo neformalno smatrati da je tip A jednostavniji od tipa B ako se oni razlikuju i tip B se gradi od A.

Pođimo sada još jedan korak dalje i razmotrimo šta će biti rezultat izraza:

```
cout << max2("niska2","niska1") << endl;
```

U ovom slučaju se radi o pokazivačima, pa će biti primenjena druga verzija šablona. Za slučaj pokazivača `const char*` naš programski kod *ne zna* da se radi o nizovima, pa poredi samo prve znakove niski i, pošto prvi znak prve niske nije veći od prvog znaka druge, rezultat će biti druga niska[35].

Ovde možemo da primenimo još jednu specijalizaciju šablona. Međutim, ako navedemo kompletan tip „`char*`“, da li nam je uopšte potreban parametar `T`? Zaista, ne samo da nam tip `T` nije potreban, već bi njegovo navođenje proizvelo suprotan efekat – zbog toga što bi kod bio ispravan za *svaki* tip `T`, implicitno instanciranje šablona funkcije bi *uvek* bilo obavljano na osnovu drugih definicija, a ne na osnovu te nove, zato što je ona *opštija*. Zbog toga pri eksplicitnoj specijalizaciji, u slučaju kada se neki parametar više ne pojavljuje u definiciji funkcije, taj parametar može (a vidimo da praktično i mora) da se izostavi:

```
template<>
char* max2( char* x, char* y ){
    return strcmp(x,y)>0 ? x : y;
}
```

Iako može da izgleda da smo dovršili posao, prevođenje dovoljno dobrim prevodiocem (tj. dovoljno saglasnim sa standardom) će proizvoditi program koji i dalje ne radi ispravno!? Problem je u tipovima niski.

Po standardu jezika, svaka eksplicitno navedena niska ima tip `const char*`, a ne `char*`. Kako konstantan tip ne može da se konvertuje u nekonstantan, ispada da naša specijalizacija ne može da se primeni u konkretnom slučaju. Rešenje je da napravimo izmenjenu verziju za konstantne tipove[36]:

```
template<>
const char* max2( const char* x, const char* y ){
    return strcmp(x,y)>0 ? x : y;
}
```

Čim se pogleda navedena specijalizacija, prirodno se postavlja pitanje – šta bi se dogodilo da smo takvu specijalizaciju napisali bez deklaracije šablona bez

[35] Da nismo napisali drugu verziju šablona, i ovde bi se, kao u slučaju pokazivača na cele brojeve, poredile adrese niski, što svejedno ne bi valjalo. Znači, drugom definicijom šablona nismo napravili ovaj problem nego samo izmenili njegov oblik.

[36] Da budemo do kraja precizni, i prethodnu opštu definiciju šablona za pokazivače bi trebalo napisati za konstantne pokazivače, zato što ne menja objekte sa kojima radi. Međutim, ona bi svejedno trebalo da radi.

parametara, tj. kao običnu funkciju? Da li bi to bilo isto ili ne? Da li bi ponašanje prevedenog programa bilo isto?

Ponašanje programa bi izvesno bilo isto, ali moramo da uočimo da specijalizacija bez parametara i obična funkcija *nisu* isto. Štaviše, možemo napisati i jedno i drugo i prevodilac neće prijaviti grešku:

```
template<>
const char* max2( const char* x, const char* y ){
    return "specijalizacija";
}

const char* max2( const char* x, const char* y ){
    return "funkcija";
}
```

Ako postoje i šablon bez parametara i funkcija sa istim imenom i tipom, prevodilac će uvek birati funkciju, osim ako je upotrebljena sintaksa eksplicitnog instanciranja šablona bez parametara. Na primer, sledeće naredbe će ispisati, redom, „funkcija“ i „specijalizacija“.

```
cout << max2("niska2","niska1") << endl;
cout << max2<>("niska2","niska1") << endl;
```

11.5 Podrazumevane vrednosti parametara

Pri definisanju šablona funkcija i klasa mogu da se upotrebljavaju i podrazumevane vrednosti parametara šablona.

Primetimo da su podrazumevane vrednosti parametara šablona funkcija dopuštene tek od standarda C++11. Ranije verzije jezika su dopuštale samo implicitno instanciranje šablona funkcija, pa podrazumevane vrednosti parametara nisu imale mnogo smisla.

Kao i u slučaju podrazumevanih vrednosti argumenata funkcija, neki parametar šablona može da ima podrazumevanu vrednost samo ako i svi parametri koji se navode iza njega takođe imaju podrazumevane vrednosti. Važno je da se pri opisivanju podrazumevanih vrednosti parametara mogu upotrebljavati čak i prethodno navedeni parametri. Ako svi parametri šablona imaju podrazumevane vrednosti, onda se eksplicitno instanciranje može izvesti i navođenjem prazne liste parametara: „`<>`“. Ako se ne navede lista parametara biće primenjeno implicitno instanciranje.

Na primer, ako definišemo šablon funkcije `max2` na sledeći način:

```
template <typename T1, typename T2=T1, typename T3=T1>
inline T3 max2( const T1& x, const T2& y )
{
```

```
    return x > y ? x : y;
}
```

onda će u izrazu `max2(1,2.5)` šablon funkcije biti instanciran sa vrednostima parametara, redom, `int`, `double` i `int`, a vrednost izraza će biti `2`, dok će u izrazu `max2<double>(1,2.5)` vrednost svih parametara biti `double` , a vrednost izraza `2.5`.

Primetimo da u ovom primeru ne smemo da upotrebimo vraćanje rezultata po referenci, zato što u slučaju da se bar jedan od tipova `T1` i `T2` razlikuje od tipa `T3` prevodilac mora da implicitno konvertuje odgovarajući argument u tip `T3`, pa dobijeni privremeni objekat neće smeti da se vrati po referenci. Posebno, ako su `T1` i `T2` različiti tipovi, onda će bar jedan od njih će biti različit od tipa `T3`, pa ćemo imati opisan slučaj. Odgovarajuća greška bi bila ispoljena tek pri instanciranju šablona sa opisanim razlikama između tipova.

Podrazumevane vrednosti šablona klasa su prisutne praktično od početnih verzija programskog jezika C++. Između ostalog, intenzivno se upotrebljavaju u standardnoj biblioteci. Šabloni klasa u standardnoj biblioteci često imaju parametre sa podrazumevanim vrednostima, koji se u uobičajenom radu veoma retko eksplicitno navode, a omogućavaju upravljanje specifičnim aspektima rada klase. Na primer, sve kolekcije (`vector`, `list`,...) imaju tipski parametar `Alloc` koji omogućava da se eksplicitno navede način upravljanja memorijom, ali se u osnovnim kursevima programskog jezika on obično i ne pominje, zato što je njegova podrazumevana vrednost u najvećem broju slučajeva dovoljno dobra:

```
template< typename T, typename Alloc = std::allocator<T> >
class vector;
```

11.6 Algoritmi i funkcionali

Praktičnu primenu šablona predstavićemo na jednom primeru *algoritma*, koji radi sa *funkcionalima*. U terminologiji C++-a, *algoritmima* se nazivaju uopštene implementacije šablona funkcija koje se upotrebljavaju za različite tipove podataka i kolekcija, koje su često parametrizovane čak i operacijama koje se u algoritmima upotrebljavaju.

Funkcionalima se, strogo posmatrano, nazivaju klase čije instance mogu da se primene kao funkcije, ali se u uobičajenoj komunikaciji funkcionalima često nazivaju i konkretne instance tih klasa, pa čak i obične funkcije. U programskom jeziku C++ funkcionali se veoma često koriste pri implementaciji uopštenih algoritama, da bi se omogućila njihova primena za što veći skup različitih tipova. Nazivaju se i *funkcijski objekti*.

Početni primer

Funkcionale ćemo predstaviti na primeru programskog koda koji prebrojava koliko elemenata neke kolekcije zadovoljava neki uslov. Početni primer ćemo napisati bez upotrebe šablona, kao uobičajeni način brojanja neparnih elemenata u nizu celih brojeva. Štaviše, u prvih nekoliko koraka nećemo koristiti funkcionale, nego obične funkcije:

```
unsigned prebrojNeparne( const vector<int>& niz )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( niz[i] % 2 )
            n++;
    return n;
}

int main(){...
    cout << prebrojNeparne(niz) << endl;
...}
```

Ovakvo rešenje je sasvim konkretno i može da se primeni samo na cele brojeve sadržane u datom nizu, a i uslov koji se proverava je fiksiran i ne može da se menja. Kroz nekoliko korak ćemo pokazati kako pomoću šablona možemo da uopštimo praktično sve aspekte problema – od tipa elemenata do tipa kolekcije i uslova koji se proverava.

Uopštavanje tipova elemenata i kolekcija

U prvom koraku ćemo samo da izdvojimo proveru uslova u posebnu funkciju, čime se u određenoj meri smanjuje spregnutost funkcije koja izvodi brojanje sa kontekstom brojanja:

```
bool neparan( int n ){
    return n%2;
}

unsigned prebrojNeparne( const vector<int>& niz )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( neparan( niz[i] ))
            n++;
    return n;
}
```

U drugom koraku ćemo da apstrahujemo uslov na način koji je uobičajen i za programski jezik C. Umesto fiksiranja uslova u samoj implementaciji brojanja, prenećemo uslov kao argument funkcije:

```
unsigned prebroj( const vector<int>& niz, bool(*uslov)(int) )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( uslov( niz[i] ))
            n++;
    return n;
}

int main(){...
    cout << prebroj(niz,neparan) << endl;
...}
```

Prva stvar koju možemo da apstrahujemo šablonom je tip elemenata kolekcije

```
template<typename T>
unsigned prebroj( const vector<T>& niz, bool(*uslov)(T) )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( uslov( niz[i] ))
            n++;
    return n;
}
```

Upotreba novog šablona je potpuno ista kao i upotreba prethodno napisane funkcije `prebroj`, zato što može da se primenjuje implicitno instanciranje šablona funkcije:

```
    cout << prebroj(niz,neparan) << endl;
```

Ako najpre promenimo način prolaska kroz kolekciju, tako da se umesto opsega indeksa upotrebljavaju iteratori, onda ćemo nakon toga moći da apstrahujemo i kolekciju. Znači, najpre iteratori:

```
template<typename T>
unsigned prebroj( const vector<T>& niz, bool(*uslov)(T) )
{
    unsigned n=0;
    typename vector<T>::const_iterator
        i = niz.begin(),
        e = niz.end();
    for( ; i!=e; i++ )
        if( uslov( *i ))
            n++;
    return n;
}
```

a zatim i apstrahovanje čitave kolekcije uopštenom kolekcijom tipa `TK`:

```
template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija, bool(*uslov)(T) )
{
    unsigned n=0;
    typename TK::const_iterator
        i = kolekcija.begin(),
        e = kolekcija.end();
    for( ; i!=e; i++ )
        if( uslov( *i ))
            n++;
    return n;
}
```

Ovako napisan metod može da izvrši brojanje odgovarajućih elemenata bilo koje kolekcije koja ima iteratore.

Primetimo da je u prethodnim primerima pri navođenju iteratora upotrebljena ključna reč `typename`. Pri upotrebi šablona prevodilac može da bude u nedoumici u vezi sa time šta programer očekuje da neko ime predstavlja u parametrizovanom tipu (u našem slučaju prevodiocu nije jasno šta u tipu `TK` predstavlja ime `const_iterator`, tj. da li je to ime tipa ili ime metoda). Imajući u vidu da imena najčešće predstavljaju podatke ili metode, prevodilac sva upotrebljena imena tako i tumači, ali to onda može da proizvede probleme ako neko ime ne predstavlja ni podatak ni metod nego ime tipa. Zbog toga, ako se u izrazu upotrebljava neki tip za koji se očekuje da bude definisan u parametrizovanoj klasi, onda takav izraz mora da bude označen ključnom reči `typename`, da bi prevodilac znao da se radi o imenu tipa.

Primetimo da od standarda C++11 postoji nova sintaksa naredbe `for`, koja omogućava obilazak cele date kolekcije, a koja se interno implementira kao obilazak pomoću iteratora, praktično na isti način kao što smo to prethodno napisali. Navodimo odgovarajući kod radi ilustracije, ali ćemo u nastavku ipak upotrebljavati verziju sa iteratorima, zato što je opštija i pruža nam neke dodatne mogućnosti:

```
template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija, bool(*uslov)(T) )
{
    unsigned n=0;
    for( auto x : kolekcija )
        if( uslov( x ))
            n++;
    return n;
}
```

Iteratori omogućavaju sekvencijalno obilaženje elemenata kolekcija, čak i kada je njihova interna struktura daleko složenija (na primer, ako predstavlja drvo, graf ili neku drugu nelinearnu strukturu). Možemo reći i da iteratori *prevode* kolekcije u

odgovarajuću sekvencijalnu reprezentaciju. Kao posledica toga se pojavljuje veoma često korišćena mogućnost da se par iteratora upotrebljava za označavanje opsega elemenata koje je potrebno da obradimo – prvi iterator određuje početak opsega (tj. njegova vrednost je iterator na prvi element opsega koji želimo da obradimo), a drugi određuje kraj opsega (tj. njegova vrednost je iterator *iza* poslednjeg elementa koji želimo da obradimo).

Ako u implementaciji šablona funkcije `prebroj` već koristimo iteratore, onda možemo da odemo i korak dalje – umesto da se ograničavamo samo na brojanje u celoj kolekciji, možemo da se bavimo i samo izabranim delovima kolekcije. Radi zadržavanja kompatibilnosti sa prethodnim verzijama, sada pišemo novu funkciju sa istim imenom, ali zadržavamo i prethodnu, s tim da je implementiramo preko nove. Parametar nove funkcije više nije tip kolekcije `TK` nego tip iteratora `Iterator`. Dodatnim primerom upotrebe ilustrovaćemo kako mogu da se prebroje svi neparni brojevi u delu niza sa indeksima od 20 do 34:

```
template<typename T, typename Iterator>
unsigned prebroj( Iterator beg, Iterator end, bool(*uslov)(T) )
{
    unsigned n=0;
    for(Iterator i=beg; i!=end; i++ )
        if( uslov( *i ))
            n++;
    return n;
}

template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija , bool(*uslov)(T) )
{
    return prebroj( kolekcija.begin(), kolekcija.end(), uslov );
}

int main(){...
    cout << prebroj(niz,neparan) << endl;
    cout << prebroj(niz.begin()+20, niz.begin()+35,
        neparan) << endl;
...}
```

Uopštavanje provere uslova

Oblik u kome smo do sada navodili funkciju zahteva da bude ispunjeno nekoliko relativno strogih uslova. Prvi je da rezultat provere uslova mora biti tipa `bool`, što nije uvek slučaj u C-u i C++-u, gde se kao rezultati logičkih funkcija često izračunavaju celi brojevi. U ovom koraku ćemo da apstrahujemo sve moguće funkcije koje smeju da se upotrebe u datom kontekstu, tj. za koje program može da se prevede i da ispravno radi. Umesto konkretnog tipa funkcije uvešćemo novi tipski

parametar `Predikat`. Zbog uvođenja novog tipskog parametra, raniji tipski parametar `T` više nije potreban, pa ćemo ga obrisati:

```
template<typename Iterator, typename Predikat>
unsigned prebroj( Iterator beg, Iterator end, Predikat uslov )
{...}

template<typename TK, typename Predikat>
unsigned prebroj( const TK& kolekcija, Predikat uslov )
{...}
```

Termin *predikat* se u programiranju veoma često upotrebljava da označi tip funkcije ili konkretnu funkciju koja izračunava logičku vrednost, tj. proverava ispunjenost nekog uslova za date podatke. U našem slučaju se radi o *unarnom* predikatu, ali to nećemo eksplicitno naglašavati u imenu tipa.

Ovako napisan šablon funkcije `prebroj` može da broji elemente bilo koje kolekcije, ili dela kolekcije, koja podržava iteratore. Da bi šablon mogao da se instancira potrebno je da kolekcija podržava iteratore i da predikat predstavlja unarnu funkciju, čiji je argument nekog tipa u koji se tip elemenata kolekcije može implicitno konvertovati. Posledica povećane fleksibilnosti je da sada za proveravanje uslova možemo da upotrebimo i funkciju koja očekuje argument tipa `double`, na primer funciju `veciOd5`:

```
bool veciOd5( double n ){
    return n > 5;
}

int main(){...
    cout << prebroj(niz,veciOd5) << endl;
    cout << prebroj(niz.begin(), niz.begin()+20, veciOd5) << endl;
...}
```

Operator ()

Na ovom mestu ćemo da razmotrimo jednu veoma zanimljivu osobinu programskog jezika C++ – programibilnost operatora „`()`“. Programski jezik C++ dopušta programerima da u klasi napišu operator „`()`“ sa proizvoljnim brojem argumenata i sa proizvoljnim tipom rezultata. Štaviše, dopušteno je da se u jednoj klasi napiše proizvoljno mnogo različitih implementacija ovog operatora, sve dok se one razlikuju po broju ili tipovima argumenata. Klase koje imaju definisan bar jedan operator „`()`“ predstavljaju *funkcionale*. Kao prvi primer funkcionala napisaćemo sasvim jednostavnu implementaciju klase `Neparan`:

```
class Neparan
{
public:
    bool operator()( int n )
```

```
        { return n%2; }
};
```

Pre nego što nastavimo dalje, uočimo specifičnu sintaksu definicije operatora „`()`“ – prvi par zagrada predstavlja deo imena operatora, dok drugi par zagrada služi za navođenje argumenata.

Objekti ove klase mogu da se upotrebljavaju kao funkcije koje imaju jedan celobrojni argument i izračunavaju logičku vrednost, na primer:

```
Neparan a;
cout << a(3) << endl;
cout << Neparan()(5) << endl;
```

gde u prvom primeru koristimo automatski objekat a kao funkciju, a u drugom primeru izrazom `Neparan()` pravimo privremeni neimenovani objekat i koristimo ga kao funkciju.

Možda nije sasvim očigledno koji je smisao pravljenja ovakvog funkcionala, kada je rezultat isti kao da smo napisali funkciju, s tim da je čak neophodno i malo više pisanja. Zaista, klasa `Neparan` u našem primeru ne donosi suštinski ništa novo u odnosu na do sada upotrebljavanu funkciju `neparan`. Da bismo razumeli suštinu moramo da se podsetimo da za razliku od funkcije, koja je jedna i uvek ista, klasa može da ima više instanci (objekata), koje se mogu razlikovati. U slučaju klase `Nepran` sve instance će biti praktično iste i zato se ne vidi značajan doprinos ovakvog pristupa, ali u slučaju klase koja ima neko unutrašnje stanje (tj. čiji objekti sadrže neke podatke), svaki objekat može da predstavlja posebnu funkciju, a klasa praktično predstavlja oblik *dinamičkog* šablona funkcije sa konstantnim parametrima.

Vratimo se funkciji `veciOd5`, sa kojom smo se susreli pre nekoliko pasusa, u kojoj postoji unapred određena i fiksirana konstanta 5 u odnosu na koju se porede argumenti funkcije. Ako bismo želeli da parametrizujemo tu konstantu, jedan način je da napišemo šablon funkcije:

```
template<int osnovaPoredjenja>
bool veciOd( int n ){
    return n > osnovaPoredjenja;
}
```

Takav šablon možemo da upotrebimo za pravljenje različith instanci funkcija, na primer:

```
int main(){...
    cout << prebroj(niz,veciOd<5>) << endl;
    cout << prebroj(niz,veciOd<15>) << endl;
...}
```

Šablon funkcije može da bude dobro rešenje za neke slučajeve, ali ipak ima značajno ograničenje: parametri šablona se navode, a instance šablona prave, isključivo *u vreme prevođenja programa*. Drugim rečima, šabloni omogućavaju isključivo *statičko* instanciranje, koje je definisano u tekstu programa i predvidivo u trenutku prevođenja. Nije moguće odložiti instanciranje do izvršavanja programa i uraditi, na primer, nešto kao:

```
for( int i=0; i<100; i+=5 )
    cout << i << " : " << prebroj(niz,veciOd<i>) << endl;
```

Tu stupaju na scenu funkcionali, zato što se instance funkcionala prave *dinamički*, u fazi izvršavanja programa. Odgovarajući funkcional možemo da napišemo kao klasu sa jednim podatkom:

```
class VeciOd
{
public:
    VeciOd( int n )
        : N_(n)
    {}

    bool operator()( int n )
        { return n > N_; }

private:
    int N_;
};
```

Zbog toga što ima unarni operator „`()`“, koji je implementiran na odgovarajući način, klasa `VeciOd` zadovoljava sve kriterijume potrebne da se tipski parametar `Predikat` instancira klasom `VeciOd`, a konkretni argument `uslov` objektom ove klase:

```
cout << prebroj(niz,VeciOd(5)) << endl;
```

Štaviše, sada možemo da napišemo i ono što prethodno nismo mogli pomoću šablona funkcije `veciOd`:

```
int main(){...
    for( int i=0; i<100; i+=5 )
        cout << i << " : " << prebroj(niz,VeciOd(i)) << endl;
...}
```

Na ovom primeru vidimo da funkcionali donose veoma značajan nov koncept u programiranju, a koji bez primene šablona ne bi mogao da dođe u potpunosti do izražaja. Upotrebom funkcionala i šablona omogućeno je parametrizovanje ne samo podataka nego i operacija koje se izvršavaju u nekom kontekstu – i to u fazi izvršavanja programa. Čitavi algoritmi se na taj način mogu apstrahovati i

implementirati nezavisno od konkretnih tipova podataka, ali i od konkretnih elementarnih operacija koje se u algoritmima koriste.

Funkcionali se upotrebljavaju u mnogim segmentima standardne biblioteke programskog jezika C++, uključujući algoritme za pretraživanje, uređivanje, particionisanje i trasnsformisanje kolekcija podataka i mnoge druge operacije. Radi ilustracije i daljeg upoznavanja oblasti preporučuje se da se analiziraju implementacije algoritama iz biblioteke `<algorithm>`. Na primer, jedan od algoritama te biblioteke je i šablon funkcije `count_if`, koji radi praktično isto što i naš šablon funkcije `prebroj`.

Vezivanje argumenta funkcije

Videli smo kako možemo da pišemo funkcionale i da ih koristimo u algoritmima. Nešto što predstavlja potencijalan problem je potreba da se pišu odgovarajuće klase, čak i kada već postoje implementirane funkcije za poređenje. Na primer, ako imamo neku funkciju koja proverava ispunjenost nekog odnosa između dva argumenta, moraćemo da napišemo odgovarajuću klasu da bismo mogli da dinamički instanciramo proveravanje uslova u odnosu na različite izabrane vrednosti drugog argumenta:

```
bool proveraOdnosa( T1 a, T2 b ){...}

class ProveraOdnosa {
public:
    ProveraOdnosa( T2 b )
        : B_(b)
    {}

    bool operator()( T1 a )
        { return proveraOdnosa(a,B_); }

private:
    T2 B_;
};
```

Ako malo bolje pogledamo takvu klasu, možemo da uočimo da tu postoji obrazac koji se ponavlja. Zaista, ako izdvojimo samo ono što je zajedničko za sve takve klase, onda možemo da napišemo odgovarajući šablon klase, na primer ovako:

```
template<typename Fn, typename T>
class VeziDrugiArg
{
public:
    VeziDrugiArg( Fn fn, T arg2 )
        : Fn_(fn), Arg2_(arg2)
    {}

    bool operator()( T n )
```

```
        { return Fn_(n,Arg2_); }

private:
    Fn Fn_;
    T  Arg2_;
};
```

Objekti ovog šablona, tj. njegovih instanci, sadržaće u sebi vezanu informaciju o funkciji koja se upotrebljava za proveravanje odnosa i o vrednosti koju je potrebno navoditi kao drugi argument pri proveravanju. Instance ovog šablona predstavljaju pojedinačne klase poput predstavljene klase `ProveraOdnosa`. Na primer, naredna dva izraza bi izračunavala identične funkcionale:

```
ProveraOdnosa(v2)
VeziDrugiArg<bool (*)(T1,T2), T1>(proveraOdnosa,v2)
```

Ako imamo šablon klase `VeziDrugiArg`, onda više nije potrebno da pišemo pojedinačne odgovarajuće klase, već možemo da ih instanciramo iz univerzalnog šablona. Preostaje još jedan problem, a to je neugodna sintaksa – svaki put moramo da eksplicitno navodimo tipove funkcije i argumenta, što otežava pisanje koda i daje nečitak rezultat. Srećom, i to može da se prevaziđe. Kao što smo već videli, instanciranje klase uvek zahteva da se tipovi eksplicitno navode i to ne možemo da izbegnemo, ali možemo da zaobiđemo. Iskoristićemo činjenicu da je, za razliku od šablona klasa, u slučaju šablona funkcija dopušteno implicitno instanciranje. Napravićemo šablon funkcije koji ne radi ništa drugo osim što pravi objekat odgovarajuće instance šablona klase:

```
template<typename Fn, typename T>
VeziDrugiArg<Fn,T> veziDrugiArg( Fn fn, T arg2 )
{
    return VeziDrugiArg<Fn,T>( fn, arg2 );
}
```

Sada ovaj šablon funkcije možemo da upotrebimo bez navođenja tipova i da napravimo isti funkcional kao u prethodnim slučajevima:

```
ProveraOdnosa(v2)
VeziDrugiArg<bool (*)(T1,T2), T1>(proveraOdnosa,v2)
veziDrugiArg( proveraOdnosa, v2 )
```

Primetimo da naši šabloni `VeziDrugiArg` i `veziDrugiArg` nisu potpuno uopšteni. Oni podrazumevaju da funkcija za poređenje ima dva argumenta koji su istog tipa, kao i da je rezultat obavezno logičkog tipa. Dalje uopštavanje bi zahtevao mnogo više prostora nego što nam je na raspolaganju, pa ga ovde nećemo opisivati.

U standardnoj biblioteci `<functional>` implementirano je više različitih operacija nad funkcionalima, među kojima i vezivanja argumenata funkcija. Ako bismo

koristili delove te biblioteke, onda bismo odgovarajući funkcional (bez ograničenja koje ima naše rešenje) mogli da napravimo ovako:

```
bind2nd( ptr_fun(proveraOdnosa), v2 )
```

a u C++-u 11 i ovako:

```
bind( proveraOdnosa, _2, v2 )
```

11.7 Upravljanje ponašanjem klase

Šabloni mogu da se upotrebljavaju za efikasno upravljanje ponašanjem objekata ciljne klase. Kada se za upravljanje ponašanjem koriste neki parametri koji se dinamički proveravaju (na primer podaci sadržani u objektu, a zadati prilikom pravljenja objekta), onda se svaki put pri njihovom proveravanju troši procesorsko vreme i gubi se na performansama. Ako se način ponašanja objekata ne menja tokom njihovog života, onda je potencijalno efikasnije da se ponašanje odredi statički, u programskom kodu, kao i da se proveravanje uskladi i optimizuje sa statički podešenim parametrima. Upravo takvo ponašanje pružaju nam šabloni klasa.

Najpre ćemo da predstavimo primenu šablona za upravljanje ponašanjem klase na primeru implementacije nizova koji omogućavaju proveravanje ispravnosti opsega indeksa, a kasnije ćemo pokušati da uočimo opštija pravila. Za početak ćemo napraviti nadgradnju bibliotečkog šablona klase `vector` tako da obuhvati proveravanje ispravnosti opsega indeksa[37]:

```
template<typename T>
class Niz : public vector<T>
{
public:
    Niz()
        : vector<T>()
    {}

    Niz( unsigned int sz )
        : vector<T>( sz )
    {}

    T& operator[]( unsigned i )
    {
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
        return vector<T>::operator[](i);
```

[37] Javnim nasleđivanjem nasledili smo i sve javne metode šablona klase. Da bi naša klasa bila jednako funkcionalna trebalo bi dodati još nekoliko konstruktora.

```
    }

    const T& operator[]( unsigned i ) const
    {
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
        return vector<T>::operator[](i);
    }
};
```

U prethodnom primeru smo napisali dve implementacije operatora „`[]`“. Razlog je u tome što želimo da ovaj operator koristimo na različite načine za konstantne i nekonstantne nizove. Ako niz nije konstantan, onda operator mora da vrati referencu na element niza, da bismo izrazom „`niz[i]=x`“ mogli da promenimo vrednost elementa niza. Sa druge strane, ako je niz konstantan, onda operator ne sme da vrati takvu referencu na element niza, zato što bismo tako posredno menjali sadržaj konstantnog niza. Rešenje je u pisanju dve implementacije[38] operatora „`[]`“.

Ponašanje ovako napisanog niza možemo da proverimo na sledeći način:

```
int main()
{
    try {
        Niz<int> niz(20);

        for( unsigned i=0; i<niz.size(); i++ )
            niz[i] = i*i;
        for( unsigned i=0; i<=niz.size(); i++ )
            cout << i << " : " << niz[i] << endl;

    }catch( exception& e ){
        cerr << "*** GRESKA: " << e.what() << endl;
    }

    return 0;
}
```

Kolekcije u standardnoj biblioteci, uključujući i klasu `vector`, ne proveravaju ispravnost indeksa, zato što je primarni cilj efikasnost, a pretpostavlja se da će programeri *znati šta rade*. Sada imamo našu verziju, koja može da proverava

[38] Na prvi pogled ove dve implementacije imaju isti broj i tip argumenata, ali zapravo nije tako. Svaki metod klase, pored svih eksplicitno navedenih argumenata, ima i jedan implicitan – pokazivač `this`, koji pokazuje na objekat na kome se metod izvršava. U navedenim verzijama operatora pokazivač `this` ima različite tipove: u prvom slučaju pokazivač `this` ima tip `Niz<T>*`, a u drugom, konstantnom slučaju, ima tip `const Niz<T>*`. Zbog toga na osnovu konteksta, tj. na osnovu očekivanog tipa objekta na kome se operator izračunava, može da se odredi koja od dve verzije implementacije će da se koristi.

efikasnost, ali je sigurno da će biti manje efikasna nego `vector`. Bilo bi idealno kada bi jedna ista klasa (ili šablon) mogla da se upotrebljava i sa proverom i bez provere indeksa, a zavisno od potreba programera, na primer tako da se tokom razvoja indeksi proveravaju, a da se u konačnoj verziji programa, koja mora da bude efikasnija, indeksi ne proveravaju, zato što se pretpostavlja da je program dovoljno dobro testiran.

Ako bi se prilagođavanje ponašanja odvijalo u fazi izvršavanja programa, onda bi to zahtevalo neki vid proveravanja, poput:

```
T& operator[]( unsigned i )
{
    if( ... potrebnoProveravatiOpsegIndeksa ... )
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
    return vector<T>::operator[](i);
}
```

U slučaju takve implementacije bi najpre moralo da se svaki put proverava „da li je potrebno proveravati opseg“, pa tek onda eventualno i da se proverava da li je opseg ispravan. Zbog toga bi u oba slučaja rešenje bilo sporije nego bez te opcije. Jedini način da se ovo unapredi je da se o načinu izvršavanja operatora odlučuje pre njegovog izračunavanja, tj. ili u fazi pravljenja objekta ili u fazi prevođenja programa. Odlučivanje u fazi pravljenja objekta bi se svodilo na dinamičko vezivanje metoda, što je već umereno sporije od statički vezanog operatora, dok bi odlučivanje u fazi prevođenja programa moglo da ponudi najbolje moguće performanse. U idealnom slučaju se deo koda koji proverava indekse uopšte ne bi ugrađivao u izvršni kod ako provere nisu potrebne.

Proveru indeksa i izbacivanje izuzetka možemo da izdvojimo u posebnu funkciju, na primer ovako:

```
T& operator[]( unsigned i )
{
    ::ProveraIndeksa( i, vector<T>::size() );
    return vector<T>::operator[](i);
}
```

Međutim, uobičajeno rešenje je da se umesto posebnog metoda napravi posebna klasa `ProveraIndeksa`, koja ima odgovarajući statički metod `Provera`. Takav pristup zahteva malo više pisanja, ali nam pruža i veće mogućnosti:

```
T& operator[]( unsigned i )
{
    ProveraIndeksa::Provera( i, vector<T>::size() );
    return vector<T>::operator[](i);
}
```

U našem primeru je dovoljno da klasa `ProveraIndeksa` ima samo metod `Provera`, koji definiše specifično ponašanje našeg niza.

```
class ProveraIndeksa {
public:
    static void Provera( unsigned indeks, unsigned opseg ){
        if( indeks >= opseg )
            throw out_of_range("Indeks van opsega");
    }
};
```

Izvedena izmena predstavlja vid refaktorisanja – nismo promenili ponašanje našeg niza, već samo strukturu koda. Indeksi se još uvek bezuslovno proveravaju, ali sada se već nazire način apstrahovanja problema. Ako pored klase `ProveraIndeksa` napišemo i klasu `BezProvereIndeksa`, a koja ima isti interfejs ali ne radi ništa, onda možemo da navođenjem jedne od ovih klasa kao dodatnog parametra šablona niza, odredimo ponašanje niza u odnosu na proveravanje indeksa i to statički, u fazi prevođenja programa.

U klasi `BezProvereIndeksa` metod `Provera` ne radi doslovno ništa:

```
class BezProvereIndeksa{
public:
    static void Provera( unsigned indeks, unsigned opseg ) {}
};
```

Šablonu `Niz` dodajemo novi tipski parametar `ProveravacIndeksa`. Proveravanje indeksa prepuštamo metodu `Provera` upotrebljenog konkretnog tipa koji se navede kao vrednost ovog novog tipskog parametra:

```
template<typename T, typename ProveravacIndeksa>
class Niz : public vector<T>
{
public:
    T& operator[]( unsigned i )
    {
        ProveravacIndeksa::Provera( i, vector<T>::size() );
        return vector<T>::operator[](i);
    }

    const T& operator[]( unsigned i ) const
    {
        ProveravacIndeksa::Provera( i, vector<T>::size() );
        return vector<T>::operator[](i);
    }
};
```

Ako se kao `ProveravacIndeksa` upotrebi klasa `ProveraIndeksa`, onda će se proveravati ispravnost opsega indeksa, a ako se upotrebi klasa `BezProvereIndeksa`, onda će se operator indeksnog pristupa izvršavati bez proveravanja ispravnosti indeksa. Štaviše, ako se provera ne vrši, zbog toga što se šabloni prevode za svaku instancu posebno, i naš operator i cela klasa `Niz` će se prevesti doslovno kao da smo upotrebljavali `vector` – bez ikakve cene po performanse[39].

Ako se neki oblik ponašanja koristi češće nego drugi, onda ono može da se navede kao podrazumevano ponašanje putem navođenja odgovarajuće podrazumevane vrednosti tipskog parametra. U našem slučaju, možemo da navedemo de je podrazumevano da ne želimo da proveravamo opseg indeksa:

```
template<typename T,
         typename ProveravacIndeksa=BezProvereIndeksa>
class Niz : public vector<T>
{...};
```

11.8 Višedimenzione matrice

Kao složeniji primer primene šablona navešćemo primer implementacije višedimenzionih „matrica“. Na ovom primeru ćemo predstaviti rekurzivnu definiciju šablona. Pod višedimenzionom matricom podrazumevamo kolekciju podataka koja se indeksira po više dimenzija i koristi nalik na višedimenzione nizove programskog jezika C, ali uz potencijalno veće mogućnosti. Koristimo naziv *matrica* a ne *niz* da bi u tekstu bilo jasnije kada govorimo o pastrahovanoj višedimenzionoj strukturi a kada o jednodimenzionom nizu, koji koristimo pri implementaciji.

Implementacija bi trebalo da nam omogući upotrebu nalik na naredni primer:

```
Matrica<int,3> m;
...
m[i][j][k] = ...;
...
```

Implementacija višedimenzione matrice zavisi od tipa elementa i broja dimenzija. Zbog toga ćemo da je implementiramo kao šablon klase sa tipskim parametrom `T` i neoznačenim celobrojnim parametrom `Dim`:

[39] Ovde moramo da istaknemo da zbog različitog pristupa prevođenju i optimizovanju programa, može da se desi da u slučaju verzija koje su prilagođene za debagovanje u izvršnoj verziji programa ostanu neki delovi „suvišnog“ koda, kako bi lakše mogao da se prati tok izvršavanja programa. Sa druge strane, optimizovana produkciona izvšna verzija bi trebalo da bude „u bajt“ identična kao u slučaju upotrebe klase `vector`.

```
template<typename T, unsigned Dim>
class Matrica {...}
```

Matricu ćemo da predstavimo kao niz matrica niže dimenzije. Matrica niže dimenzije će imati tip `Matrica<T,Dim-1>`. To će ujedno biti i tip elemenata niza kojim implementiramo matricu:

```
template<typename T, unsigned Dim>
class Matrica
{
public:
    typedef Matrica<T,Dim-1> tPodmatrica;
...
private:
    vector<tPodmatrica> _Podmatrice;
};
```

Ovakvo rešenje počiva na primeni rekurzije po parametru šablona. Kao i svaka druga rekurzija, i ova mora da ima završni korak, koji je eksplicitno definisan. U slučaju šablona klasa, završni korak se definiše eksplicitnom specijalizacijom završnog slučaja. Završni slučaj se definiše za vrednosti parametara za koje neki od elemenata rekurzivnog šablona nije ispravan. U slučaju naše matrice razlikovanje nastupa u slučaju dimenzije 1, pa zato moramo da napravimo eksplicitnu specijalizaciju za taj slučaj:

```
template<typename T>
class Matrica<T,1>
{
...
private:
    vector<T> _Elementi;
};
```

Sve potrebne metode moramo da razvijamo paralelno – kako za opštiji slučaj, tako i za specijalan slučaj dimenzije 1. Najpre ćemo da napišemo metod `PostaviVelicinu` za postavljanje dimenzija matrice. Pretpostavićemo da je argument funkcije pokazivač na niz neoznačenih celih brojeva, koji redom predstavljaju dimenizije matrice. Pri postavljanju veličine moramo da promenimo veličinu niza podmatrica, a zatim da svakoj od njih na odgovarajući način postavimo novu veličinu navođenjem sufiksa niza dimenzija `dims+1`. Dodaćemo i metod `Velicina` koji izračunava veličinu odgovarajuće dimenzije matrice:

```
template<typename T, unsigned Dim>
class Matrica { ...

    void PostaviVelicinu( unsigned dims[] ) {
         _Podmatrice.resize(dims[0]);
        for( unsigned i=0; i<dims[0]; i++ )
```

```
            _Podmatrice[i].PostaviVelicinu(dims+1);
    }

    unsigned Velicina( unsigned d ) const {
        return d
            ? _Podmatrice[0].Velicina(d-1)
            : _Podmatrice.size();
    }

... };
```

Isto moramo da uradimo i za specijalan slučaj dimenzije 1:

```
template<typename T>
class Matrica<T,1> { ...

    void PostaviVelicinu( unsigned dims[] ) {
        _Elementi.resize(dims[0]);
    }

    unsigned Velicina( unsigned d ) const {
        return _Elementi.size();
    }

... };
```

Preostaje da napišemo metode za pristupanje elementima matrice. Upotrebićemo odgovarajuće implementacije operatora „`[]`”:

```
template<typename T, unsigned Dim>
class Matrica { ...

    tPodmatrica& operator[]( unsigned i ) {
        return _Podmatrice[i];
    }

    const tPodmatrica& operator[]( unsigned i ) const {
        return _Podmatrice[i];
    }

... };


template<typename T>
class Matrica<T,1> { ...

    T& operator[]( unsigned i ) {
        return _Elementi[i];
    }

    const T& operator[]( unsigned i ) const {
        return _Elementi[i];
    }
```

```
... };
```

Radi ilustracije upotrebe, popunićemo matricu nekim vrednostima i zatim ispisati njen sadržaj:

```
int main()
{
    Matrica<int,3> m;
    unsigned dimenzije[] = {2,3,4};
    m.PostaviVelicinu( dimenzije );
    cout << m.Velicina(0) << endl;
    cout << m.Velicina(1) << endl;
    cout << m.Velicina(2) << endl;

    for( unsigned i=0; i<m.Velicina(0); i++)
        for( unsigned j=0; j<m.Velicina(1); j++ )
            for( unsigned k=0; k<m.Velicina(2); k++ )
                m[i][j][k] = i*100 + j*10 + k;

    for( unsigned i=0; i<m.Velicina(0); i++){
        for( unsigned j=0; j<m.Velicina(1); j++ ){
            for( unsigned k=0; k<m.Velicina(2); k++ )
                cout << m[i][j][k] << ' ';
            cout << "  ";
        }
        cout << endl;
    }

    return 0;
}
```

Primetimo da je inicijalizacija veličine matrice izvedena na nezgodan način. Bilo bi mnogo jednostavnije kada bi to moglo da se uradi navođenjem dimenzija u konstuktoru:

```
Matrica<int,3> m(2,3,4);
```

Tom prilikom bi se već pri prevođenju mogao prepoznati neispravan broj argumenata, dok bi u prethodnom slučaju to izazvalo grešku tek u radu i to sa nepredvidivim posledicama. Problem je što ne znamo unapred koliko dimenzija ima matrica i koliko argumenata može da ima konstruktor.

To se rešava tzv. variadičkim šablonima, gde se broj parametara šablona ostavlja da bude promenljiv, tj. da bude prilagodljiv potrebama i kontekstu. Variadički parametar šablona se označava sa tri tačke: „`typename... Args`“ a koristi takođe sa tri tačke „`Args...`“. U našem slučaju, za početak bi trebalo da napišemo metod `PostaviVelicinu` tako radi sa odgovarajućim brojem argumenata, a zatim da tome prilagodimo i konstruktore:

```
template<typename T, unsigned Dim>
class Matrica { ...

    template<typename... Args>
    Matrica( unsigned n, Args... args )
    {
        PostaviVelicinu( n, args... );
    }

    template<typename... Args>
    void PostaviVelicinu( unsigned dim, Args... args ) {
        _Elementi.resize( dim );
        for( auto& m: _Elementi )
            m.PostaviVelicinu( args... );
    }

... };

template<typename T>
class Matrica<T,1> { ...

    Matrica( unsigned n  )
    {
        PostaviVelicinu( n );
    }

    void PostaviVelicinu( unsigned dim ) {
        _Elementi.resize( dim );
    }

... };
```

Primetimo da smo na ovaj način u šablonskoj klasi definisali još i šablonski konstruktor. Iako može da izgleda da je to prilično fleksibilno i nedefinisano, zapravo će u svakoj konkretnoj instanci matrice moći da se koristi samo jedan ovakav konstruktor i to upravo onaj koji nam je potreban – sa istim brojem argumenata kao što je dimenzija matrice, i to tako da svi budu tipa `unsigned`.

Ovakav šablon matrice je upotrebljiv, ali ima još mnogo prostora za unapređenja. Na primer, u implementaciji bismo umesto kolekcije `vector` mogli da koristimo prethodno napisan šablon klase `Niz`, koji može da proverava ispravnost indeksa.

Drugo unapređenje se odnosi na organizaciju memorije. Opisana implementacija matrice se svodi na pravljenje velikog broja malih objekata. Na primer, ako napravimo matricu dimenzija 10x20x30, onda bi se „velika“ matrica sastojala od 10 manjih matrica dimenzija 20x30, a svaka od njih od po 20 nizova sa po 30 elemenata. Alokacija memorije za takvu matricu bi podrazumevala alokaciju 211 nizova, što predstavlja neefikasan oblik organizacije memorije.

Alternativa je da se svi podaci koji čine matricu alociraju u jednom koraku, čime bi bili bolje locirani i efikasniji za alokaciju i upotrebu (između ostalog, takva organizacija je bolje prilagođena keširanju). Takvo rešenje, sa druge strane, zahteva nešto složeniji način implementiranja pristupanja elementima matrice. U prethodnom primeru prva primena operatora indeksiranja je vraćala referencu na matricu niže dimenzije, koja fizički postoji u implementaciji matrice, ali u alternativnom slučaju takva matrica ne postoji fizički kao poseban objekat i zato ove operatore moramo da implementiramo drugačije. Jedan način implementacije je da se operatori indeksiranja naprave tako da izračunavaju objekte slične iteratorima, koji bi se takođe implementirali kao šabloni.

Preporučujemo čitaocima da implementiraju oba opisana unapređenja, kao dobre primere za vežbanje.

11.9 Lambda izrazi i funkcionali

Lambda izrazi nisu neposredno vezani za parametarski polimorfizam, ali se veoma često upotrebljavaju zajedno sa šablonima, a i implementiraju se pomoću funkcionala i kao zamena za funkcionale, kojima smo već posvetili pažnju u ovom poglavlju, pa ćemo im zato posvetiti pažnju na ovom mestu.

U prethodnim odeljcima smo predstavili rad sa funkcionalima ali smo se zadržali na dobrim stranama te tehnike. Međutim, ona ima i neke slabosti. U praktičnom radu se najviše ispoljavaju dva osnovna problema.

Prvo, ako se neka funkcija (ili funkcional) upotrebljava na samo jednom mestu u programu, onda se njenim definisanjem možda nepotrebno opterećuje prostor imena? Šta se dešava ako su za implementaciju našeg programa potrebne stotine takvih funkcija?

Drugi problem je možda još važniji – navedeni način implementiranja primorava programera da fizički razdvoji ponašanje konkretne funkcije od mesta na kome se ona upotrebljava. Time se potencijalno otežava razumevanje programa, a time i njegovo održavanje. Značaj takve razdvojenosti se dodatno povećava ako je to jedino mesto upotrebe, pa još i ako se radi o relativno jednostavnim funkcijama.

Ako pogledamo način definisanja funkcionala, možemo da uočimo da svi imaju tipski oblik: definišemo koje vrednosti iz opsega primene želimo da vežemo, a zatim definišemo telo funkcije koja se izračunava. Stiče se utisak da bi to možda moglo da se uradi i tako da sve navodimo na istom mestu, bar nalik na:

```
cout << prebroj(niz, ... n%2 ... ) << endl;
```

Programski jezik C++, od verzije 11, ima mogućnost definisanja i korišćenja umetnutih neimenovanih pomoćnih funkcija, koje se uobičajeno nazivaju *lambda funkcije* ili *lambda izrazi*. Osnovna namena lambda izraza je upravo prevazilaženje

navedenih problema pri korišćenju funkcija i funkcionala. Lambda izrazi apstrahuju opisane sličnosti između svih funkcionala i omogućavaju da se na mestu upotrebe navede programski kod koji se kasnije ponaša praktično ekvivalentno kao da su upotrebljeni funkcionali. Lambda izrazi omogućavaju da se u programskom kodu, na mestu na kome se očekuje da se navede referenca na neku ranije definisanu funkciju ili na neki funkcijski objekat (funkcional), umesto toga navede upravo definicija funkcije. U pitanju je tehnika koja je uobičajena za funkcionalne programske jezike, a njen naziv potiče iz lambda računa[40].

Sintaksa lambda izraza u programskom jeziku C++ ima sledeći osnovni oblik:

```
[ <vezani objekti> ] ( <argumenti> ) { <telo> }
```

Argumenti predstavljaju argumente funkcije, a telo predstavlja telo funkcije i navode se na potpuno isti način kao i u slučaju uobičajenog definisanja imenovanih funkcija. Ono što se razlikuje je što (1) ne navodimo naziv funkcije, (2) najčešće ne moramo da navodimo tip rezultata funkcije i (3) možemo da navedemo dodatno vezivanje objekata. Puna sintaksa (od C++20) omogućava i definisanje šablonskih lambda izraza, koji imaju određene specifične primene u šablonima, ali to ovde nećemo detaljnije razmatrati.

Na primer, i Ispravna definicija lambda izraza koji proverava da li je neoznačen ceo broj neparan može da bude napisana ovako:

```
[]( unsigned n ){ return n%2; }
```

a njegova primena pri prebrojavanju bi izgledala ovako:

```
cout << prebroj(niz, [](unsigned n){return n%2;} ) << endl;
```

Naziv funkcije se ne navodi, zato što nije potreban – da jeste, pisali bismo klasičnu funkciju (ili funkcional) a ne lambda izraz. Alternativno, ime može da se uvede i kao ime promenljive funkcijskog tipa, na primer:

```
auto neparan = []( unsigned n ){ return n%2; };
```

[40] Lambda račun predstavlja jedan od formalnih načina za definisanje pojma algoritma. Oblikovao ga je Alonso Čerč 1930-ih godina, u cilju formalizacije rukovanja matematičkim izrazima. Jedan od osnovnih elemenata lambda računa je apstrakcija unarne funkcije, tzv. lambda izraz. Na primer, ako bismo imali na raspolaganju odgovarajuće operatore, onda bi se sintaksom lambda izraza funkcija neparan deifnisala kao: `λx.x%2!=0` [Church1936].

Tip rezultata funkcije obično može da se ustanovi automatski, pa zato ne mora da se navodi. Ako tip funkcije ne može da se automatski nedvosmisleno ustanovi, onda se navodi u sledećem obliku:

```
[ <vezani objekti> ] -> <tip rezultata> ( <argumenti> ) { <telo> }
```

Vezivanje objekata

Vezivanje objekata omogućava da se definišu lambda izrazi u čijem telu se ne koriste samo argumenti nego i neka ili sva imena (tj. objekti) koja su vidljiva u okvirima u kojima je definisan lambda izraz. Ako se ne navede nijedan objekat niti parametar vezivanja, onda u telu lambda izraza ne može da se koristi nijedan drugi objekat osim onih koji se navode kao argumenti.

U prethodnim primerima smo pravili funkcional `VeciOd`. Za razliku od tog primera, upotreba lambda izraza sa vezivanjem podataka nam daje mnogo kompaktniji programski kod. Umesto da pišemo celu klasu, sada možemo da napišemo samo lambda izraz u kome ćemo naglasiti da je potrebno da se veže objekat `i`, tako što ćemo navesti `i` u spisku vezanih objekata:

```
for( int i=0; i<100; i+=5 )
    cout << i << " : "
         << prebroj(niz,[i](int n){return n>i;}) << endl;
```

Kao što vidimo iz primera, da bi objekat mogao da se koristi u telu lambda izraza, potrebno je da se njegovo ime navede u spisku vezanih objekata. Podrazumevano je da se vezivanje objekata odvija kopiranjem. Ako želimo da vezani objekat može da se menja u telu lambda izraza, onda je potrebno da se veže po referenci. To se postiže tako što se u spisku vezanih imena ispred imena objekta navede „`&`“.

Zapravo, imajući u vidu veličinu objekata, obično ima smisla da se vezivanje kopiranjem zameni vezivanjem po referenci na konstantan objekat, ali moramo imati u vidu da to nije uvek moguće. Na primer, ako se napravljen lambda izraz vraća kao rezultat funkcije, a vezani su neki privremeni objekti, onda je jasno da tu mora da se izvodi kopiranje, zato što bismo inače koristili reference na objekte koji su u međuvremenu obrisani. To je posebno važno imati na umu i kada se radi o većim objektima, zato što neoprezna upotreba vezivanja objekata po vrednosti može da naruši performanse.

Kao što su lambda izrazi bez vezanih objekata konceptualno ekvivalentni funkcijama, iz navedenog primera može da se vidi da su lambda izrazi sa vezivanjem objekata konceptualno ekvivalentni funkcionalima. Zaista, sve što

možemo da implementiramo pomoću funkcionala, možemo da implementiramo i pomoću lambda izraza sa vezivanjem objekata i obrnuto[41].

U spisku vezanih objekata može da se navede i „`this`“, čime se postiže da je u telu lambde na raspolaganju pokazivač `this`, t.j. objekat u čijem se metodu pravi lambda izraz. Za `this` je ponašanje drugačije od uobičajenog – podrazumevano je da se objekat prenosi po referenci i da može da se menja (što je i logično, jer se radi o prenošenju pokazivača a ne objekta). Ako želimo da se ne menja (tj. da se vezuje kao kopija) onda u spisku vezanih objekata umesto `this` mora da stoji `*this`, pri čemu je to moguće tek od C++17.

Ako želimo da se vezuju sva raspoloživa imena, onda navodimo „`[=]`“. Tada će sva imena (koja se koriste u telu lambda izraza) biti vezana po vrednosti, kao kopije. Ako želimo da sva imena budu vezana po referenci, onda navodimo „`[&]`“. Ako se navede „`[=]`“, onda i dalje za neke objekte može da se naglasi da se vezuju po referenci, na primer „`[=,&x]`“.

Od verzije C++14 uz vezivanje može da se navede i inicijalizacija, što praktično znači da se pravi i inicijalizuje novi objekat sa datim imenom. To je isto kao da se neposredno pre definicije lambda izraza navede odgovarajuća definicija objekta.

Prevođenje lambda izraza

Lamda izrazi koji ne koriste vezivanje imena mogu da se prevedu i obično se prevode kao obične funkcije. U postupku optimizacije takve funkcije mogu da se integrišu u kod funkcije iz koje se pozivaju, ali to ne menja suštinu – upotreba lambda izraza umesto običnih definisanih imenovanih funkcija ne donosi značajne razlike u odnosu na veličinu i efikasnost izvršnog programa. Teorijski ne bi trebalo da bude nikakvih razlika, ali praktično ipak može da bude manjih odstupanja, posebno ako se prevođenje radi u režimu za debagovanje.

Kada je reč o lambda izrazima koji koriste vezivanje objekata, već smo ranije ukazali na konceptualnu ekvivalentnost takvih izraza sa funkcijskim objektima. Zaista, takvi lambda izrazi se i prevode kao funkcionali. Svi vezani objekti (ako se zaista i koriste u telu lambda izraza) se implementiraju kao članovi podaci funkcijskih objekata. Ni u ovom slučaju ne bi trebalo da bude razlika u veličini i efikasnosti izvršnog programa, bez obzira na to da li u programu eksplicitno definišemo funkcionale ili koristimo lambda izraze sa vezivanjem objekata.

---

[41] Naravno, to stoji samo ako pod funkcionalima podrazumevamo klase koje od ponašanja imaju samo konstruktor i operator (). Neki složeniji oblici takvih funkcionala, koji pri računanju menjaju i svoje unutrašnje stanje, nisu mogli da se opišu lambda izrazima u C++11. Međutim od uvođenja inicijalizacije vezanih objekata, tj. od verzije C++14. čak je i to moguće. Ako funkcionali rade i neke druge stvari i imaju i neke druge metode, onda je jasno da ne mogu da se implementiraju lambda izrazima.

std::function

Kada pišemo neku funkciju (tj. šablon funkcije ili klase) koja može da ima za argument neku funkciju, lambda izraz ili funkcional, u nekim slučajevima u postupku prevođenja može da bude problema oko automatskog ustanovljavanja tipova. Problemi nastaju, pre svega, ako pokušavamo da sačuvamo dobijene funkcijske parametre, ali i ako želimo da kao rezultat funkcije vratimo neku lambda funkciju.

Radi rešavanja tog problema u standardnoj biblioteci je definisan šablon klase `std::function`, koji omogućava da se ujednači rad sa svim vrstama „funkcija“, tj. svih objekata na koje može da se primeni operator aplikacije „`()`“ odgovarajućeg tipa. Parametar šablona je suštinski tip funkcije, a ne stvarni tip objekta sa kojim radimo. Suštinskim tipom funkcije smatramo osnovni tip funkcije poput koje se naš tip ponaša. Na primer, ako pravimo funkcional koji se ponaša kao funkcija koja preslikava dva cela broja u logičku vrednost, onda je suštinski tip funkcije za taj funkcional upravo `bool(int,int)`.

U sledećem primeru pravimo funkcional koji poredi dati broj sa 5 i čuvamo ga pod imenom `veciOd5`. Suštinski tip tog funkcionala je `bool(int)`:

```
int n=5;
std::function<bool(int)> veciOd5 = [n](int x){ return x>5; };
```

11.10 ...Koncepti

Rad sa šablonima može da bude neugodan zato što prevođenje zna da dugo traje i poruke o greškama zahtevaju pažljivo čitanje i razumevanje. Uzrok većine grešaka sa šablonima je što se pokušava njivoha primena na neke konkretne tipove za koje iz nekog raloga ne rade. Pri tome iz deklaracije šablona, posebno ako su složeni, nije očigledno koje uslove moraju da zadovolje neki tipovi da bi mogli da se koriste pri instanciranju tipskih parametara. Koncepti su uvedeni upravo da bi se taj problem rešio.

Koncepti u programskom jeziku C++ imaju sličnu ulogu kao klase tipova u Haskelu, na primer. Služe da opišemo neke klase tipova i da zatim pri definisanju šablona ne navodimo opšte tipske parametre, nego da označimo da parametar mora da pripada nekoj opisanoj klasi tipova, tj. *da zadovoljava dati koncept*, prema terminima programskog jezika C++.

Koncept se definiše u obliku:

```
template<...>
concept ime = izraz;
```

gde `ime` predstavlja ime koncepta a `izraz` je logički izraz koji navedeni tipski parametri moraju da zadovolje da bi bili smatrani delom tog koncepta. Na primer, koncept `IzvedenaIz<A,B>` će važiti ako je klasa `A` izvedena iz klase `B`:

```
template< class A, class B >
concept BaznaZa = std::is_base_of<B,A>::value;
```

Nakon što se napravi, koncept može da se koristi pri pravljenju šablona. Može da se navodi odmah iza klauzule deklarisanja tipskih parametara, ili na kraju dela deklaracije, a pre definicije. Koriste se u obliku klauzule `requires` i izraza `requires`. Klauzula `requires` ima sledeći oblik:

```
requires izraz
```

gde je `izraz` neki konstantan logički izraz (izračunljiv u fazi prevođenja) i predstavlja uslov koji mora da bude ispunjen.

Na primer:

```
template<class T1, class T2> requires IzvedenaIz<T1,T2>
...
```

naglašava da klasa `T1` mora da bude izvedena iz `T2`.

Izraz `require` se koristi u jednom od dva oblika:

```
requires { izraz }
requires ( argumenti ) { izraz }
```

gde `izraz` predstavlja neki izraz (ili naredbu ili niz naredbi) programskog jezika, eventualno praćen deklaracijom tipova nekih imena koja se pojavljuju u izrazu, i naglašava da taj izraz mora da bude prevodiv za konkretne tipove. Izraz `requires` može da se koristi i pri definisanju koncepata.

Naredni primer definiše koncept `ImaSabiranje` koji obuhvata sve tipove koji imaju operator sabiranja:

```
template<class T>
concept ImaSabiranje requires ( T x ){ x + x; };
```

Uslovi koji se definišu, pa tako i odgovarajući koncepti, mogu biti relativno složeni. Standardna biblioteka je zbog toga dopunjena velikim brojem unapred pripremeljenih koncepata i još većim broje šablonskih predikata (šablona klasa koji imaju statičku logičku vrednost `value` izračunljivu u toku prevođenja, kao na primer `std::is_base_of<B,A>`).

11.11 Generički tipovi u Javi i C#-u

Pod uticajem programskog jezika C++ i tehnika programiranja u nekim funkcionalnim programskim jezicima, nekim savremenim programskim jezicima je naknadno dodata podrška za mehanizme koji liče na šablone programskog jezika C++.

Programski jezik Java od verzije 5 ima podršku za generičke tipove (ili *generičke parametre*) (engl. *generics*). Generički tipovi u Javi su razvijeni u velikoj meri po uzoru na šablone u programskom jeziku C++, ali se od njih suštinski razlikuju. Dok šabloni u programskom jeziku C++ predstavljaju implementaciju suštinskog parametarskog polimorfizma, generički tipovi u Javi imaju pre svega sintaksnu namenu, kako bi se jedan značajan deo grešaka u kodu uočio već u fazi prevođenja programa, umesto u fazi njegovog izvršavanja.

Da bismo razumeli kontekst, podsetimo se da su kolekcije i mnoge druge stvari u Javi izvorno radile samo sa uopštenom osnovnom klasom `Object`, pri čemu je često bilo neophodno da se vrši eksplicitna konverzija tipova. Na primer, ovakav program bi se uspešno preveo, ali bi proizveo grešku pri izvršavanju, zato što bi došlo do neuspešnog pokušaja konverzije niske u ceo broj:

```
List v = new ArrayList();
v.add("niska");
Integer i = (Integer)v.get(0);   // greška u fazi izvršavanja
```

Osnovni cilj uvođenja generičkih tipova je bio da se spreče takvi problemi, tako što bi se omogućilo da se u kodu eksplicitno navode tipovi elemenata kolekcija, kao i da zatim ne moraju da se eksplicitno navode konverzije tih elemenata:

```
List<String> v = new ArrayList<String>();
v.add("niska");
Integer i = v.get(0);            // greška u fazi prevođenja
```

Zbog toga se generički tipovi ponašaju značajno drugačije od šablona i imaju mnogo manje mogućnosti. Navešćemo neke od najvažnijih razlika:

- Generički tipovi u Javi nisu uvedeni sa ciljem omogućavanja punog generičkog polimorfizma, već prvenstveno sa ciljem olakšavanja pisanja koda i prevencije određenih vrsta grešaka;
- U programskom jeziku C++ šabloni se praktično prevode tek pri instanciranju i sam parametarski tip se ne koristi za ozbiljnije sintaksne i semantičke provere, dok se generički tipovi u Javi proveravaju i koriste pre svega u fazi prevođenja;

- Da bi uspelo prevođenje generičkog koda u Javi, ako se na generičkim tipovima primenjuju neki metodi, onda mora biti naznačeno koja su postojeća ograničenja (tj. implementirani interfejsi ili nasleđene klase) za te generičke tipove;
- Prevođenje generičkog Java koda se odvija uz tzv. *brisanje tipova*. Prevod generičkog koda će biti potpuno isti kao da je svuda upotrebljena uopštena klasa `Object` ili druga odgovarajuća bazna klasa koja je navedena kao ograničenje, uz odgovarajuće eksplicitne konverzije[42];
- Svaka instanca šablona u C++-u se prevodi posebno i proizvodi poseban prevod, tako da se i statički podaci šablona klasa u C++-u razlikuju za svaku konkretnu instancu. Sa druge strane, generičkom tipu u Javi odgovara samo jedan deljeni prevod pa se i statički podaci dele za sve primene generičkog tipa. Štaviše, zbog toga statički podaci u Javi ne smeju imati generički tip;
- Prevod šablona u C++-u će biti drugačiji i posebno optimizovan za svaki konkretan tip, dok se generički kod u Javi ni na koji način niti posebno prevodi niti posebno optimizuje za konkretan tip;
- U Javi se instanciranje može obavljati samo klasama (naslednicima uopštene klase `Object`), dok se u C++-u mogu upotrebljavati proizvoljni tipovi;
- U Javi ne postoje konstantni parametri, već samo tipski.

Kao što vidimo, iako ima sintaksnih sličnosti, suštinske razlike su zapravo mnogo veće, pa se sa pravom može reći da generički tipovi u Javi predstavljaju pre sintaksnu olakšicu nego punu implementaciju parametarskog polimorfizma.

Slično tome, programskom jeziku C# je u verziji 2 dodata podrška za generičke tipove. U osnovi se radi o sličnom konceptu kao u slučaju Jave. Sintaksa je veoma slična, a razlike se uglavnom odnose na način prevođenja. U slučaju programskog jezika C# podrška za generičke tipove je izvedena na nivou virtualne mašine, a ne samo na nivou prevodioca. Zbog toga se prevođenjem dobija različit kod za različite instance generičkih tipova, što omogućava sprovođenje određenih optimizacija. Ipak, i pored toga generički tipovi u jeziku C# su mnogo bliži istoimenom konceptu kod Jave nego šablonima u jeziku C++. Neke od najvažnijih razlika u odnosu na C++ su:

[42] Korisna posledica ovakvog pristupa je da se prevodi generičkog koda pisanog na Javi 5 mogu upotrebljavati u programima pisanim na Javi 4, naravno, ako osim generičkih tipova nisu korišćene druge specifičnosti Jave 5.

- Kao i u slučaju Jave, generički tipovi u C#-u dopuštaju samo tipske parametre;
- Nije podržana eksplicitna ni parcijalna specijalizacija;
- Tipski parametar ne može da se koristi kao bazna klasa generičke klase;
- Ne postoje podrazumevane vrednosti parametara;
- U slučaju C#-a programski kod mora da može da se ispravno prevede za sve vrednosti tipskih parametara koje zadovoljavaju zadato ograničenje.