Dizajn programskih jezika

Programski jezik C++

Milena Vujošević Janičić

www.matf.bg.ac.rs/~milena

Dizajn programskih jezika
Beograd, 29. novembar, 2016.

Pregled

Pregled

Polimorfizam

- Polimorfizam — puno formi — različita ponašanja
- Izraz ili vrednost je polimorfan ako može da ima za domen različite tipove

Polimorfizam

Na primer,

$$a < b$$

Polimorfizam

Na primer,

$$a < b$$

- Ako poredimo dva objekta, ta dva objekta mogu da budu nekakvi brojevi (prirodni, celi, realni), ali mogu da budu i niske karaktera ili nešto treće

Polimorfizam

Na primer,

$$a + b$$

Polimorfizam

Na primer,

$$a + b$$

- Ako sabiramo dva objekta, ta dva objekta mogu da budu nekakvi brojevi (prirodni, celi, realni), ali mogu da budu i razlomci, kompleksni brojevi, niske karaktera ili nešto treće

Polimorfizam

- Polimorfizam je ključan faktor ponovne iskoristivosti koda.
- Postoje različiti oblici polimorfizma, najznačajniji su hijerarhijski i parametarski polimorfizam
- Postoje još i implicitni i ad-hoc polimorfizam
- Podela na polimorfizme može da bude i na statičke (parametarski i ad-hoc) i dinamičke polimorfizme (hijerarhijski i implicitni)

Hijerarhijski polimorfizam

- Hijerarhijski polimorfizam počiva na konceptu nasleđivanja klasa u objektno orijentisanim jezicima.
- Kod napisan da radi sa objektima jedne klase može da radi i sa objektima svih njenih potklasa
- Hijerarhijski polimorfizam omogućava da se poziv metoda iz odgovarajuće klase razreši dinamički (tj u fazi izvršavanja programa).

Hijerarhijski polimorfizam

- Na primer, ukoliko u klasi `Vozilo` postoji metod `BrojTockova` onda se taj metod može primeniti i nad objektima klasa `Automobil`, `Kamion`, `Avion`... ukoliko su te klase implementirane kao potklase klase `Vozilo`

Primer

```
class Vozilo
{
public:
    Vozilo() {}
    virtual ~Vozilo() {}
    virtual int BrojVrata() const = 0;
    virtual int BrojTockova() const = 0;
    virtual int BrojSedista() const = 0;
};
```

Primer

```
class Automobil : public Vozilo
{
public:
    int BrojVrata() const
        { return 4; }

    int BrojTockova() const
        { return 4; }

    int BrojSedista() const
        { return 4; }
};
```

Primer

```
class Kamion : public Vozilo
{
public:
    int BrojVrata() const
        { return 2; }

    int BrojTockova() const
        { return 6; }

    int BrojSedista() const
        { return 3; }
};
```

Primer

```
int main() {
    int vrsta;
    Vozilo* v;
    cin >> vrsta;
    if(vrsta==1) v = new Automobil();
    else v = new Kamion();
    cout << "Osobine vozila su:" << endl;
    cout << "Broj vrata: " << v->BrojVrata() << endl;
    cout << "Broj tockova: " << v->BrojTockova() << endl;
    cout << "Broj sedista: " << v->BrojSedista() << endl;
    return 0;
}
```

Implictni i ad-hoc polimorfizam

- Implicitni polimorfizam je karakterističan za funkcionalne programske jezike i o tome ćemo više pričati u okviru funkcionalne paradigme.
- Implicitni polimorfizam dozvoljava pisanje koda bez navođenja tipova. Tipovi se određuju u fazi izvršavanja.
- Implicitni polimorfizam je uopštenje parametarskog polimorfizma.

Implictni i ad-hoc polimorfizam

- Specifičan vid implicitnog polimorfizma su makroi u C-u
  `#define kub(x) (x)*(x)*(x)`
  Makro `kub` se može upotrebiti za svaki tip podataka za koji je definisan operator `*` (npr. `int`, `float`, `double`, `signed int`, `long int` ...)
- S druge strane, ukoliko bi želeli da napišemo funkciju `kub`, bilo bi potrebno da je pišemo za svaki pojedinačni tip:
  ```
  double kubD(double x);
  int kubI(int x);
  float kubF(float x);
  ..
  ```
  ili da se oslonimo na svojstva implicitnih konverzija, npr.
  `double kub(double x)`

Implictni i ad-hoc polimorfizam

- Ad-hoc polimorfizam se oslanja na mogućnost zadavanja istog imena funkcijama koje se razlikuju po tipu i/ili broju argumenata (preopterećivanje — eng. overloading)
- Takođe se može smatrati specifičnom vrstom parametarskog polimorfizma.

Primer

```
int      kub(int    x);   umesto int      kubI(int      x);
float    kub(float  x);           float    kubF(float    x);
double   kub(double x);           double   kubD(double   x);
Razlomak kub(Razlomak x);         Razlomak kubR(Razlomak x);

ili

int saberi(int x, int y);
int saberi(int x, int y, int z);
int saberi(int x, int y, int z, int w);

          umesto

int saberi2(int x, int y);
int saberi3(int x, int y, int z);
int saberi4(int x, int y, int z, int w);
```

Parametarski polimorfizam i generičko programiranje

- Generičko programiranje je stil programiranja u kojem se algoritmi pišu tako da se tip podataka nad kojim se algoritam primenjuje apstrahuje
- Generičko programiranje se prvi put pojavljuje 1973. godine u jeziku ML — pisanje funkcija koje se mogu primeniti na različite tipove

Parametarski polimorfizam i generičko programiranje

- U osnovi generičkog programiranja nalazi se koncept parametarskog polimorfizma
- Parametarski polimorfizam predstavlja mehanizam koji čini jezik ekspresivnijim, pri čemu se zadržavaju svojstva statičke provere tipova.

Motivacija

- Stroga tipiziranost ima kao posledicu da se i jednostavne funkcije moraju definisati više puta da bi se mogle upotrebljavati nad raznim tipovima.
- Na primer,
  - Funkcija `kub`

```
int      kub(int x)      { return x*x*x; }
double   kub(double x)   { return x*x*x; }
Razlomak kub(Razlomak x) { return x*x*x; }
```

  - Funkcija koja računa minimum dva prirodna broja, dva realna broja, dva razlomka...

```
int      minimum(int x, int y)           { return x<y ? x:y; }
double   minimum(double x, double y)     { return x<y ? x:y; }
Razlomak minimum(Razlomak x, Razlomak y) { return x<y ? x:y; }
```

Motivacija

- To dovodi do redudantnosti u pisanju koda, želimo da imamo jednu funkciju koja će umeti da sračuna kub i jednu funkciju koja računa minimum bez obzira na tip podataka nad kojim se vrednost računa

Motivacija

Gde još možemo da primenimo ovu ideju?

Motivacija

Gde još možemo da primenimo ovu ideju?

- Par, niz, lista (jednostruko ili dvostrko povezana), red, stek, skup, mape...
- Minimum, maksimum
- Sortiranje, linerna pretraga, binarna pretraga
- Jednakost delova kolekcije, umetanje, brisanje, zamena
- Obrtanje, rotiranje, mešanje
- ...

Motivacija

- U okviru parametarskog polimorfizma koriste se tipske promenljive
- Da bi se upotrebio kod koji koristi tipske promenljive potrebno je tipskoj promenljivoj dodeliti nekakav konkretan tip za koji se kod može ispravno prevesti i izvršiti

Motivacija

Na primer,

- Da bi se izračunao kub, potrebno je definisati operator * — za svaki tip za koji postoji definisan ovaj operator, isti algoritam za računanje kuba treba da može da se upotrebi
- Da bi se izračunao minimum potrebno je definisati kriterijum poređenja — za svaki tip za koji postoji definisan kriterijum poređenja, isti algoritam za računanje minimuma treba da može da se upotrebi

Generičko programiranje i programski jezici

- Programski jezik C++ podržava i ohrabruje korišćenje parametarskog polimorfizma
- Standardna biblioteka C++-a se u potpunosti zasniva na ovom konceptu
- Koncept šablona klasa i funkcija je osnova parametarskog polimorfizma u C++-u
- Paramtarski polimorfizam javlja se, u slabijoj formi, i u okviru programskih jezika Java i C#

Pregled

Programski jezik C++

- C++ je programski jezik opšte namene
- Jako popularan jezik koji se koristi za razvijanje najrazličitijih aplikacija
- Ima imperatvine, objektno orijentisane i generičke karakteristike
- Najnoviji standard za C++ uvodi i podršku lambda izrazima (funkcionalnom programiranju)
- C++ pored naprednih koncepata, ima i podršku za direktan rad sa memorijom
- Kroz C++ ilustrovaćemo koncepte generičkog programiranja

Kratka istorija

- 1979 — C sa klasama, Bjarne Stroustrup (danski naučnik, radio u Bell Labs)
- Ideja: unapređivanje jezika C sa konceptima iz jezika Simula
- 1983 — C sa klasama preimenovan u C++, po operatoru inkrementiranja
- 1985 — The C++ Programming Language — osnovna referenca do pojave standarada
- 1989 — Druga verzija, praćena novim izdanjem knjige

Standardi

- C++ se standardizuje od strane ISO (International Organization for Standardization)
- 1998 — prvi standard (ISO/IEC 14882:1998)
- 2003 — C++03, ISO/IEC 14882:2003
- 2007 — C++07, ISO/IEC TR 19768:2007
- 2011 — C++11, ISO/IEC 14882:2011
- 2014 — C++14, ISO/IEC 14882:2014
- 2017 —
- C++ je imao veliki uticaj na druge programske jezike, pre svega na jezike JAVA, C# kao i na novije verzije C-a

Uvođenje koncepata

- U okviru C-a sa klasama su uvedene klase, nasleđivanje, strogo tipiziranje, podrazumevani argumenti, inline-funckije
- 1983 — virtualne funkcije, preopterećivanje funkcija i operatora, reference, konstante, poboljšana kontrola sa tipovima, bezbedniji rad sa memorjom, komentar oblika //
- 1989 — C++2.0 — višestruko nasleđivanje, apstraktne klase, statičke metode, zaštićene metode
- Dodavanje šablona, izuzetaka, prostora imena, konverzija, logički tip, lambda izrazi...
- Podrška velikom broju naprednih koncepata, ali i velika efikasnost

Odnos C-a i C++-a

- C++ nasleđuje dosta sintakse iz C-a
- C++ se smatrao nadgradnjom i nadskupom C-a, ali to nije slučaj u potpunosti
- Iako se većina koda napisanog u C-u može komplirati sa C++ prevodiocem, to ipak nije uvek slučaj
- Na primer, C++ uvodi nove ključne reči (npr class, new, delete) koje se u C programima mogu koristiti za imena promenljivih
- C++ ima strožiju kontrolu tipova i ne dozvoljava neke implicitne konverzije koje su dozvoljene u C-u
- S druge strane, C je uveo neke ključne reči i koncepte koji nisu podržani u C++-u

Zdravo

Prevođenje: g++ program.cpp
Pokretanje: ./a.out

```
#include <iostream>

int main()
{
   std::cout << "Zdravo!\n";
   return 0; //nije neophodno, za main se podrazumeva
}
```

Osnovni elementi

- Prostor imena
- Reference
- Klase
- Nivoi vidljivosti: javna, zaštićena, privatna
- Konstruktori, destruktor
- Operatori
- Nasleđivanje
- ...

Prostor imena

- Prostori imena se definišu radi prevazilaženja problema zagađivanja globalnog prostora imena.
- Do zagađivanja dolazi pri upotrebi velikog broja različitih bibilioteka i pri pisanju velikih programa.
- Zagađivanje ima više neugodnih posledica: isto ime može da se upotrebljava u više biblioteka, ili greškom može da se navede pogrešno ime koje postoji u nekoj drugoj biblioteci
- U C-u je taj problem prevazilažen dodavanjem dugačkih identifikatora biblioteka koji otežavaju čitljivost

Prostor imena

- Prostor imena omogućava da se imena organizuju u manje celine, tj da se imena koja čine neku veću celinu definišu u posebnom prostoru imena.
- Prostor imena čine sva imena definisana u nekom bloku tog prostora imena
- Jedan prostor imena se može definisati u više blokova ili modula

```
namespace <ime prostora imena> {
...
}
```

Prostor imena

- Imena definisana u jednom prostru imena se mogu upotrebljavati u tom istom prostoru imena na uobičajen način.
- Da bi se ime upotrebljavalo van tog prostra imena, potrebno je učiniti jednu od naredne tri stvari:
  - Ispred imena svaki put navoditi ime odgovarajućeg prostora imena, npr
    `...<ime prostora imena>::<ime>`
  - Najaviti da je to ime vidljivo u tekućem prostoru imena:
    `using <ime prostora imena>::ime`
  - Najaviti da su sva imena iz nekog prostora imena vidljiva u tekućem prostoru imena:
    `using namespace <ime prostora imena>`

Zdravo

```
#include <iostream>

int main()
{
    std::cout << "Zdravo!\n";
}
```

Zdravo

```
#include <iostream>

using std::cout;

int main()
{
    cout << "Zdravo!\n";
}
```

Zdravo

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Zdravo!\n";
}
```

Reference

- Rerenca je alias, tj drugo ime za već postojeći objekat
- Jednom kada se referenca inicijalizuje, za dati objekat se može koristiti bilo ime objekta, bilo njena referenca

Primer

```
int main ()
{
   int    i;
   double d;    // deklaracija običnih promenljivih

   int    &r_i = i;
   double &r_d = d;   // deklaracija referenci

   i = 5;
   cout << "Vrednost promenljive i : " << i << endl;
   cout << "Vrednsot reference : " << r_i  << endl;

   r_d = 11.7;
   cout << "Vrednost promenljive d : " << d << endl;
   cout << "Vrednost reference : " << r_d  << endl;

   return 0;
}
```

Reference

- Reference se naješće koriste kao argumenti ili povratne vrednosti funkcija na onim mestima gde biste u C-u koristili pokazivače
- Reference omogućavaju sintaksu koja nas oslobađa upotrebe $*$ i ->

C++

```
void swap (int &a, int &b)
{
  int pom = a;
  a = b;
  b = pom;
}
```

C

```
void swap (int *a, int *b)
{
    int pom = *a;
    *a = *b;
    *b = pom;
}
```

Reference

- Reference se interno najčešće i implementiraju preko pokazivača
- Razlika između referenci i pokazivača je
  - Referenca ne može da bude vezana za NULL, pokazivač može da ima vrednost NULL
  - Referenca mora da bude inicijalizovana kada se kreira, pokazivači mogu da se inicijalizuju bilo kad.
  - Jednom kada se referenca inicijalizuje nekim objektom, ona se ne može menjati da pokazuje na drugi objekat. Pokazivači mogu da menjaju na koga ukazuju u svakom trenutku.

Klase

- Definicija klase

```
class ime
{
//Funkcije clanice
//...
//Clanovi podaci
//...
};
```

- Deklaracija klase

```
class ime;
```

Nivoi vidljivosti: javna, zaštićena, privatna

- Javna vidljivost (javni članovi su raspoloživi svim korisnicima klase): ključna reč `public`
- Zaštićena vidljivost (zaštićeni članovi su raspoloživi samo unutar definicija metoda klase i njenih naslednika): ključna reč `protected`
- Privanta vidljivost (privatni članovi su na raspolaganju samo unutar definicija metoda klase): ključna reč `private`

Primer

```
class Razlomak
{
public:
  int Brojilac() const
  { return _Brojilac; }
  int Imenilac() const
  { return _Imenilac; }
private:
 int _Brojilac;
 int _Imenilac;
};
```

Konstruktori

- Prilikom kreiranja objekta poziva se metod koji se naziva konstruktor
- Konstruktor vrši inicijalizaciju objekta pri njegovom kreiranju
- Konstruktor ima isto ime kao klasa i nema povratnu vrednost
- Jedna klasa može da ima više konstruktora koji se razlikuju po broju ili tipovima argumenata

Konstruktori

- Podrazumevani konstruktor nema argumenata i ima prazno telo metoda — automatski se generiše samo ukoliko se ne navede ni jedan drugi konstruktor
- Podrazumevani konstruktor obezbeđuje da se svi podaci objekta inicijalizuju korišćenjem odgovarajućih konstruktora bez argumenata

Primer

```
class Razlomak
{
public:
   Razlomak (int b, int i) {
    _Brojilac = b;
    _Imenilac = i;
   }
   Razlomak (int b) {
    _Brojilac = b;
    _Imenilac = 1;
   }
   Razlomak () {
    _Brojilac = 0;
    _Imenilac = 1;
   } ...
```

Podrazumevane vrednosti argumenata

```
class Razlomak
{
public:
   Razlomak (int b=0, int i=1) {
    _Brojilac = b;
    _Imenilac = i;
   }
...
```

Lista inicijalizacija

```
class Razlomak
{
public:
   Razlomak (int b=0, int i=1) :
     _Brojilac(b), _Imenilac(i)
     {}
...
```

Primer upotrebe

```
int main() {
  Razlomak a(1,2), b(3), c;
  cout << a.Brojilac() << "/" << a.Imenilac() << endl;
  cout << b.Brojilac() << "/" << b.Imenilac() << endl;
  cout << c.Brojilac() << "/" << c.Imenilac() << endl;
}
```

Preopterećivanje operatora

- Operator sabiranja

```
Razlomak operator + ( const Razlomak& y ) const {
  return Razlomak(
     Brojilac() * y.Imenilac() + Imenilac() * y.Brojilac(),
     Imenilac() * y.Imenilac()
     );
}
```

- `x + y`

  $\Leftrightarrow$

  `x.operator+(y);`

Operatori

- Operator poređenja

```
bool operator < ( const Razlomak& x ) const {
  return Brojilac() * x.Imenilac() < Imenilac() * x.Brojilac();
}
```

- `x < y`

  $\Leftrightarrow$

  `x.operator<(y);`

Destruktor

- Svaki objekat na kraju svog života mora da počisti sve ono što nakon njegovog uništavanja više nije potrebno
- Destruktor obezbeđuje deinicijalizaciju objekta pri njegovom uklanjanju
- Podrazumevani destruktor se ponaša inverzno podrazumevanom konstruktoru: obezbeđuje da svi podaci objekta budu deinicijalizovani primenom destruktora za odgovarajuće tipove

Destruktor

- Destruktor ima isto ime kao klasa i prefiks ~, nema argumenata niti povratnu vrednost
- U prethodnom jednostavnom primeru destruktor nije potreban
- Destruktori su neophodni kada se u toku života objekta alociraju neki resursi koje je zatim neophodno osloboditi, npr dinamički napravljeni objekti, dodatna memorija, otvorene datoteke...
- Za dinamičko kreiranje objekata koristi se izraz `new`, a za uklanjanje `delete`
- Direktna manipulacija sa memorijom daje efikasnost u izvršavanju programa, ali i velike mogućnosti za pravljenje grešaka

Primer

```
class Niz {
  public:
    Niz(int n) {
      _Elementi = new int[n];
      _Duzina = n;
    }
...
    ~Niz() {
      delete [] _Elementi;
    }
  private:
    int* _Elementi;
    int _Duzina;
}
```

Konstruktori

- Plitko i duboko kopiranje
- Konstruktor kopije

```
Niz(const Niz& n) :
   _duzina(n._duzina),
   _elementi( n.Duzina()>0 ? new int[n.Duzina()] : nullptr )
{
  for( unsigned i=0; i<_duzina; i++ )
     _elementi[i] = n.Element(i);
}
```

- Operator dodele

Za vežbu

- Napisti klasu Datum
- Napisati klasu Kompleksan broj
- Dovršiti klasu Niz
- Napisati klasu Lista
- ...

Nasleđivanje

- Nasleđivanje je jedan od osnovnih mehanizama u C++ i jedan od osnovnih koncepata objektno orijentisanog programiranja
- Nasleđivanje omogućava da se nova klasa opiše uz pomoć neke postojeće klase.
- Nova klasa će preuzeti sve što joj odgovara iz stare klase i promeniti ili dopuniti preostalo.
- Nasleđivanje omogućava korišćenje već napisanog koda na jednostavan i prirodan način.

Nasleđivanje

- Sintaksa nasleđivanja

  `class imeKlase: lista_izvodjenja_klase`
- Lista izvođenja klase predstavlja niz klasa koje ova klasa nasleđuje sa opisom načina tog nasleđivanja, dakle

  `vrsta_nasledjivanja ime_klase_koja_se_nasledjuje`

  Elementi liste su razdvojeni zarezima.
- Vrste nasleđivanja mogu biti **private, protected i public.**

Primer

```
class A
{...};
class B: public A
{...};
class C: protected B
{...};
class D
{...};
class E: public A, private D
{...};
```

Nasleđivanje

- Klasa koja nasleđuje neku drugu klasu naziva se **izvedena** klasa ili **podklasa**.
- Klasa koju ta klasa nasleđuje je njena **bazna** klasa ili nadklasa.
- Ako je A bazna klasa za B, a B bazna klasa za C onda kažemo da je A **posredna bazna klasa** za C.
- Bazne i izvedene klase formiraju **hijerarhiju klasa**.
- C++ podržava **višestruko** i **jednostruko** nasleđivanje.
- Nasleđivanje je izuzetno važan koncept o kojem ćemo detaljno pričati u okviru kursa Programske paradigme kada budemo proučavali objektno orijentisanu paradigmu

Parametarski polimorfizam u C++ — Šabloni funkcija

- Šablon — template
- Pomoću šablona se pišu polimorfne funkcije i klase
- Polimorfizam funkcija se ostvaruje apstrahovanjem tipova ili konstati koje se koriste u okviru funkcije
- To su najčešće tipovi ili konstante u okviru argumenata ili rezultata, ali mogu se apstrahovti i tipovi i konstante koji se koriste u samoj implementaciji

Parametri šablona

- Najpre je potrebno deklarisati parametre šablona, a zatim i sam šablon
- Na početku definicije ili deklaracije šablona uvek stoji ključna reč `template`.
- Iza ove ključne reči uvek stoji lista parametara šablona koji su međusobno odvojeni zarezima, a koja se navodi između simbola < i >.
- Ova lista naziva se **lista parametara šablona**.

```
template<lista parametara sablona>
```

Parametri šablona

- Lista parametara šablona sastoji se od vrste parametra i imena parametra:

```
template<vrsta_parametra1 ime_parametra1,
...
vrsta_parametraN ime_parametraN>
```

- Parametar može biti tipski (ključna reč `typename`, ranije je korišćena reč `class`) Na primer

```
template<typename T>
template<typename T1, typename T2>
```

- Parametar može biti konstantni (ključna reč je tip konstante)

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

Definicija šablona

- Nakon deklaracije šablona, može se navesti deklaracija ili definicija funkcije
- Šablonske funkcije se razlikuju od običnih funkcija po tome što mogu da koriste parametrizovane tipove i konstante, npr da na mestima na kojima bi obična funkcija koristila neki konkretni tip, šablonska funkcija može da koristi tipsku promenljivu

Primer

```
int kub(int x) {
  return x*x*x;
}

template <typename T>
T kub(T x) {
  return x*x*x;
}
```

Obično se šabloni pišu na sledeći način:

Primer

```
int kub(int x) {
  return x*x*x;
}

template <typename T>
T kub(T x) {
  return x*x*x;
}
```

Obično se šabloni pišu na sledeći način:

```
template <typename T>
T kub(const T& x) {
  return x*x*x;
}
```

Primer

```
int minimum(int x, int y) {
return x<y ? x:y;
}
```

Primer

```
int minimum(int x, int y) {
return x<y ? x:y;
}

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

Instanciranje šablona

- Ukoliko želimo da koristimo prethodno napisane šablone, potrebno je da se njihovi tipovi konkretizuju
- Ovaj postupak se naziva instanciranje šablona i moguće je implicitno i eksplicitno instanciranje šablona
- Eksplicitno instanciranje podrazumeva da programer navede konkretne tipove i konstante koje želi da se koriste (ukoliko se navede tip podataka za koji nisu podržani svi odgovarajući operatori, onda kompajler prijavljuje grešku)
- Implicitno instanciranje podrazumeva da kompajler sam zaključi koji tipovi treba da se koriste
- Funkcijski šabloni se ne prevode. Za svaki konkretan upotrebljeni tip prevođenje se izvodi posebno.

Primer

```
int a,b;
a = kub<int>(5); //primer eksplicitnog instanciranja
b = kub(5);      //primer implicitnog instanciranja
```

Implicitno instanciranje šablona moguće je samo kada se jednoznačno mogu utvrditi vrednosti parametra šablona

```
int a,b;
a = minimum<int>(5,7.6);//primer eksplicitnog instanciranja
                         //uz implicitnu konverziju
b = minimum(5, 7.6);   //greska - nije moguce
                       //implicitno instanciranje
```

Primer

```
int minimum(int x, int y) {
return x<y ? x:y;
}

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

Šta nedostaje?

Primer

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

Šta nedostaje?

Primer

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

Šta nedostaje?
Kakvo ponašanje šablona želimo za `minimum(&x, &y)`?

Primer

```
// sablon za pokazivace
template <typename T>
const T* minimum( const T* x, const T* y ) {
    return *x < *y ? x : y;
}
```

Šta nedostaje?

Primer

```
// sablon za pokazivace
template <typename T>
const T* minimum( const T* x, const T* y ) {
    return *x < *y ? x : y;
}
```

Šta nedostaje?
Kakvo ponašanje želimo za
`minimum("informatika", "racunarstvo")`

Primer

```
// mozemo napisati i neparametrizovan sablon
template <>
const char* minimum( const char* x, const char* y ) {
    return strcmp(x,y)<0 ? x : y;
}
```

Primer

```
// mozemo napisati i neparametrizovan sablon
template <>
const char* minimum( const char* x, const char* y ) {
    return strcmp(x,y)<0 ? x : y;
}

// mozemo napisati i konkretnu implementaciju
const char* minimum( const char* x, const char* y ) {
    return strcmp(x,y)<0 ? x : y;
}
```

Kako se određuje koja će funkcija biti pozvana za `minimum("informatika", "racunarstvo");`

Pravila

- Ukoliko postoji konkretna funkcija (tj koja nije šablon) i koja može da se primeni, onda će ona biti najpre izabrana
- Ukoliko postoji više verzija šablona koje mogu da se primene u nekoj konkretnoj situaciji, kompajler bira onu čija je deklaracija najmanje opšta
- Ukoliko kompajler ne može da prepozna čija deklaracija je najmanje opšta, prijaviće grešku

Primer konstantnog parametra

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

//primer konstantnog parametra
template <typename T, int velicina>
T* napraviNiz() {
    return new T[velicina];
}

main() {
    // pravimo niz znakova duzine 200
    char* niska = napraviNiz<char,200>();
    int*  niz   = napraviNiz<int,100>();
    ...
    delete [] niz;
    delete [] niska;
    return 0;
}
```

Primer konstantnog parametra

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

template <int n, typename T>
void ispis( T x ) {
    for( int i=0; i<n; i++ )
        cout << x << ' ';
    cout << '\b';
}
```

```
main() {

    ispis<5,int>(2);
    ispis<3,char>('w');
    cout << endl;

    return 0;
}
```

Primer upotrebe podrazumevanih vrednosti

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

template <typename T>
void ispisIntervala
        ( T x, T y, T korak=1 )
{
    for( T i=x; i<=y; i+=korak )
        cout << i << ' ';
    cout << '\b';
}
```

```
main() {
    //Ispisuje interval za karaktere
    ispisIntervala( 'a', 'e', (char)2 );
    cout << endl;

    //Puna sintaksa, nije potrebno vrsiti
    //konverziju dvojke u char
    ispisIntervala<char>( 'a', 'e', 2 );
    cout << endl;

    ispisIntervala( 23.4, 27.8, 0.2 );
    cout << endl;

    //Koristi se podrazumevana
    //vrednost za korak
    ispisIntervala( 23, 27);

    return 0;
}
```

Podrazumevane vrednosti parametara

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

int a,b;
a = minimum<int>(5,7.6);//primer eksplicitnog instanciranja
                         //uz implicitnu konverziju
b = minimum(5, 7.6);    //greska - nije moguce
                        //implicitno instanciranje
```

Podrazumevane vrednosti parametara

```
template <typename T1, typename T2>
T1 minimum( const T1& x, const T2& y ) {
  return x < y ? x : y;
}

//template <typename T1, typename T2>
//T2 minimum( const T1& x, const T2& y ) {
//  return x < y ? x : y;
//}

std::cout << minimum(1,0.9) << std::endl;
std::cout << minimum(0.9,1) << std::endl;
std::cout << minimum<double>(1,0.9) << std::endl;
std::cout << minimum<double,double>(1,0.9) << std::endl;
```

Podrazumevane vrednosti parametara

```
g++ template.cpp -std=c++11
```

```
#include <iostream>

template <typename T1, typename T2=T1, typename T3=T1>
T3 minimum( const T1& x, const T2& y ) {
  return x < y ? x : y;
}
int main() {
  std::cout << minimum(1,0.9) << std::endl;
  std::cout << minimum(0.9,1) << std::endl;

  std::cout << minimum<double>(1,0.9) << std::endl;
  std::cout << minimum<double, double, double>(1.6,1.8) << std::endl;

  std::cout << minimum<double, int, double>(0.9,1) << std::endl;
  std::cout << minimum<double, double, int>(1.6,1.8) << std::endl;
  std::cout << minimum<int,int,double>(1.6,1.8) << std::endl;
}
```

Parametarski polimorfizam u C++ — Šabloni klasa

- Mehanizam šablona jezika C++ omogućava generisanje klasnih tipova, tj pomoću šablona klasa generišu se polimorfni klasni tipovi podataka

```
template <lista parametara sablona>
class Klasa1 { ... };
```

- Kada se jednom deklariše, tipski parametar služi kao specifikator tipa za ostatak definicije šablona klase.
- Unutar definicije šablona klase on može da se upotrebi na sasvim isti način kao što bi mogao neki ugrađeni ili korisnički definisan tip u definiciji nešablonske klase.
- Na primer, tipski parametar može da se koristi za deklarisanje podataka članova, funkcija članica, članova ugneždenih klasa itd.

Primer

Definicija šablona klase određuje kako se konstruišu pojedine klase kada je dat skup od jednog ili više stvarnih tipova ili vrednosti.

```
//Definicija sablona
template <typename T>
class Klasa1 { ... };

//konkretizacija sablona
Klasa1<int> ki;
Klasa1<double> kd;
Klasa1<Razlomak> kr;
```

Za razliku od funkcijskih šablona, instanciranje šablona klase mora da bude eksplicitno

Klasa par

```
class Par {
public:
    Par( int p, int d )
        : _prvi(p), _drugi(d)
        {}
    int prvi() const
        { return _prvi; }
    int drugi() const
        { return _drugi; }

private:
    int _prvi;
    int _drugi;
};
```

Upotreba

```
Par p(4,5);
```

Šablon klase par

```
template <typename T>
class Par {
public:
    Par( const T& p, const T& d )
        : _prvi(p), _drugi(d)
        {}
    const T& prvi() const
        { return _prvi; }
    const T& drugi() const
        { return _drugi; }
private:
    T _prvi;
    T _drugi;
};
```

Upotreba

```
Par<int> p(4,5);
Par<char> p1('a','b');
Par< Par<double> > p2( Par<double>(1.1, 2.2),
                       Par<double>(3.3,4.4));
```

Primer - podrazumevana vrednost konstantnog parametra

```
//Definicija sablona
template <typename T, int velicina = 256>
class Niz { ... };

//konkretizacij sablona
Niz<double, 200> n1; //T=double, velicina = 200
Niz<Razlomak> n2;    //T=Razlomak, velicina = 256
```

Primer - podrazmevana vrednost tipskog parametra

```
//Definicija sablona
template <int velicina, typename T = int>
class Niz { ... };

//konkretizacij sablona
Niz<200, double> n1;  //T=double, velicina = 200
Niz<256> n2;          //T=int, velicina = 256
```

Pregled

STL — Standard Template Library

- STL je veoma doprinela širokoj promenjivosti i popularnosti jezika C++
- Ova biblioteka nije bogata kao biblioteka Jave, ali se odlikuje visokim nivoom apstrakcije
- Biblioteka je strogo tipizirana, tj tipovi se proveravaju u fazi prevođenja a ne u fazi izvršavanja
- Veoma efikasna — elementi biblioteke se posebno prevode za svaki konkretan tip podataka za koji se upotrebljavaju

STL

- Biblioteka obuhvata više različitih celina
  - Koncepti iteratora i funkcionala
  - Kolekcije podataka
  - Algoritmi
  - Biblioteka tokova

Kolekcije podataka

- Sve kolekcije su implementiranje kao šabloni klasa
- Postoji jedinstven način obilaska kolekcije (iteratori)
- Kolekcije su strogo tipizirane — svi elementi kolekcije moraju biti istog tpa
- Vrste kolekcija: sekvencijalne, apstraktne i uređene

Sekvencijalne kolekcija

- Sekvencijalne kolekcije sadrže redom elemente nekog tipa.
- Tri osnovne sekvencijalne kolekcije koje možemo izdvojiti su
  - niz – `vector`
  - lista – `list`
  - dek – `deque`.

Uređene kolekcija

- Uređene kolekcije (asocijativne) efikasno vrše proveru prisustva kao i izdvajanje pojedinih elemenata.
- Elementi su sortirani po ključu.
- Primeri su
  - katalog – `map`
  - skup – `set`
- Ključevi mogu biti ponovljeni `multiset` i `multimap`

Apstraktne kolekcija

- Apstraktne kolekcije — imaju apstraktnu strukturu koja oslikava njihov primenu, dok interna struktura može biti predmet izbora.
- Primeri su
  - stek – `stack`
  - red – `queue`
  - red sa prioritetom – `priority_queue`

Iteratori

- Iteratori su apstrakcija pokazivača
- Iteratori omogućavaju da se na isti način obilaze različite kolekcije podataka

```
for( int i=0; i<velicina; i++ )
    cout << niz[i] << endl;
```

```
for( int i=0; i<velicina; i++ )
    cout << niz[i] << endl;

for( int i=0; i<velicina; i++ )
    cout << *(niz+i) << endl;
```

```
for( int i=0; i<velicina; i++ )
    cout << niz[i] << endl;

for( int i=0; i<velicina; i++ )
    cout << *(niz+i) << endl;

for( int* i=niz; i<niz+velicina; i++ )
    cout << *i << endl;
```

```
for( int i=0; i<velicina; i++ )
    cout << niz[i] << endl;

for( int i=0; i<velicina; i++ )
    cout << *(niz+i) << endl;

for( int* i=niz; i<niz+velicina; i++ )
    cout << *i << endl;

int* pocetak = niz;
int* kraj = niz+velicina;
for(int* i = pocetak; i!=kraj; i++)
    cout << (*i) << endl;
```

Iteratori

- U okviru svake kolekcije definiše se klasa `iterator` kao apstrakcija pokazivača na elemente kolekcije
- Iteratori imaju barem naredne metode: operatore ++, ==, !=, * (dereferenciranje), ->.
- U okviru svake apstraktne kolekcije definišu se metodi
  - `begin()` — izračunava početni iterator, koji se odnosi na prvi element kolekcije
  - `end()` — izračunava završni iterator, koji se odnosi na prvi element iza elemenata kolekcije

Iteratori

- Pred običnih operatora, postoje i konstantni iteratori koji omogućavaju samo čitanje elemenata kolekcije `const_iterator`
- Postoje i obrnuti iteratori koji omogućavaju obilazak elemenata kolekcije u suprotnom smeru `reverse_iterator` i `const_reverse_iterator` kao i odgovarajuće metode `rbegin` i `rend`

Sekvencijalna kolekcija vector

Osnovne karakteristike vektora su:

1. Nasumični pristup elementima vektora (npr. pristup 5-om pa 17-om elementu i td.) je veoma efikasan (predstavlja fiksni pomeraj u odnosu na početak vektora).
2. Umetanje elementa bilo gde osim na kraj vektora nije efikasno jer bi zahtevalo premeštanje svih elemenata desno od umetnutog za jedno mesto udesno.
3. Brisanje elementa sa bilo kog mesta osim sa kraja nije efikasno jer bi zahtevalo premeštanje svih elemenata desno od izbrisanog za jedno mesto ulevo.

Kako vektor raste?

- Da bi vektor dinamički rastao, on mora da obezbedi dodatnu memoriju za čuvanje nove sekvence, da redom iskopira elemente stare sekvence, i da oslobodi memoriju koju je zauzimala stara sekvenca.
- Ako bi se vektor povećavao posle svakog umetanja to bi bilo neefikasno (pogotovo ako su elementi vektora objekti klasa pa je pri kopiranju za svaki element potrebno pozvati konstruktor kopije a pri brisanju destruktor za taj element).
- Zato, kad se javi potreba za povećanjem vektora, obezbeđuje se dodatni kapacitet memorije koji prevazilazi trenutne potrebe vektora i čuva se u rezervi.
- Količina tog dodatnog kapaciteta definisana je kroz implementaciju.

Osobine lista

- Lista predstavlja nesusedne oblasti memorije koje su dvostruko povezane parom pokazivača koji pokazuju na prethodni i sledeći element omogućavajući pri tom istovremeno pristupanje elementima i unapred i unazad.
- Osnovne karakteristike liste su:
  1. Nasumični pristup elementima liste nije efikasan. Naime, da bi se pristupilo nekom elementu neophodno je pristupiti svim prethodnim elementima.
  2. Umetanje i brisanje elementa na bilo kom mestu u listi je efikasno. Potrebno je samo premestiti pokazivače ali elemente nije potrebno dirati.
  3. Potrebna je dodatna memorija za po dva pokazivača za svaki element liste.

Vektor ili lista?

- Pri izboru tipa sekvencijalnog kontejnera postoji nekoliko kriterijuma:
  1. Ako se zahteva nasumični pristup elementima, vektor je bolji u odnosu na listu.
  2. Ako je unapred poznat broj elemenata koje treba sačuvati, opet je poželjnije izabrati vektor.
  3. Ako je potrebno često umetati i brisati elemente na mestima različitim od kraja, bolji izbor je lista.
  4. Ukoliko je potrebno nasumično pristupati elementima ali i nasumično ih brisati i umetati, vrši se procena u odnosu na cenu nasumičnog pristupa kroz cenu nasumičnog umetanja/brisanja.

Dek

- Kao i u slučaju vektora, elementi su u okviru deka poređani po svom rednom broju - indeksu.
- Moguće je pristupanje elementima na osnovu indeksa, efikasno dodavanje na početak i kraj i njihovo uklanjanje sa početka i kraja.
- Dodavanje elemenata u sredinu kolekcije i brisanje elemenata iz sredine kolekcije zahtevaju premeštanje većeg broja drugih elemenata i predstavljaju neefikasne operacije
- Vektor, lista ili dek?

Neke metode klase vector, list i deque

- Šabloni ovih klasa su definisan i sa dva parametra: prvi je tip elemenata kolekcije, a drugi je način alociranja novih objekata i najčešće ćemo za ovaj parametar koristiti podrazumevanu vrednost

```
template< class T, class Allocator = allocator<T> >
class list;

template< class T, class Allocator = allocator<T> >
class vector;

template< class T, class Allocator = allocator<T> >
class deque;
```

Neke metode klase vector, list i deque

- Konstruktor bez argumenata

```
list()
vector()
deque()
```

- Konstruktor kolekcije date veličine - Elementi se inicijalizuju primenom konstruktora bez argumenata ili kopiranjem datog objekta x klase T

```
list( size_type n, const T& x =  T() )
vector( size_type n, const T& x =  T() )
deque( size_type n, const T& x =  T() )
```

Neke metode klase vector, list i deque

- Metodi koji vraćaju početne i završne iteratore

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Ako je objekat konstantan, i rezultat je konstantan iterator

Neke metode klase vector, list i deque

- Izračunavanje veličine kolekcije i najveće dopuštene veličine

```
size_type size() const
size_type max_size() const
```

- Promena veličine kolekcije: ako se smanjuje, višak se uklanja, ako se povećava, novi elementi se prave primenom konstruktora bez argumenata ili kopiranjem datog objekta x klase T

```
void resize (size_type n)
void resize (size_type n, const T& x)
```

Neke metode klase vector, list i deque

- Metodi za dodavanje i brisanje elementa sa kraja kolekcije. Menja se veličina kolekcije, ne sme se brisati iz prazne kolekcije.

```
void push_back (const T& x)
void pop_back()
```

- Metodi za dodavanje i brisanje sa početka kolekcije, ne postoje u klasi vector, samo u klasi list i deque. Menja se veličina kolekcije, ne sme se brisati iz prazne kolekcije.

```
void push_front (const T& x)
void pop_front()
```

Neke metode klase vector, list i deque

- Metodi za pristupanje prvom i poslednjem elementu kolekcije

```
const T& front() const
T& front()
const T& back() const
T& back()
```

- Metodi za pristup i-tom elementu kolekcije, ne postoji za listu, samo za vector i deque

```
const T& at(size_type n) const
T& at(size_type n)
const T& operator[](size_type n) const
T& operator[](size_type n)
```

Neke metode klase vector, list i deque

- `bool empty() const` - vraća true ako je kolekcija prazna, inače false.
- `void clear()` - briše sadržaj kolekcije.
- U okviru klase vector, bitne su naredne metode
  - `size_type capacity() const` - vraća kapacitet tj. broj elemenata koje je moguće dodati u vektor pre nego što on izraste.
  - `void reserve(size_type n)` - postavlja kapacitet vektora na vrednost n.

Neke metode klase vector, list i deque

- Metodi za umetanje elemenata u kolekciju (umetanje se vrši pre elementa na koji pokazuje iterator position)

```
iterator insert (iterator position, const T& x)
iterator insert (iterator position, size_type n, const T& x)
template <typename InputIterator>
  void insert (iterator position,
               InputIterator first, InputIterator last)
```

Neke metode klase vector, list i deque

- Metodi za brisanje elemenata liste, rezultat je iterator koji pokazuje a elementneposredno iza poslednjeg obrisanog ili end() ako je obrisan poslednji element liste

```
iterator erase (iterator position)
iterator erase (iterator first, iterator last)
```

- Kompletna razmena sadržaja dve kolekcije sa elementima istog tipa

```
void swap (list<T, Allocator>& v )
void swap (vector<T, Allocator>& v )
void swap (deque<T, Allocator>& v )
```

Neke metode klase vector, list i deque

- Definisani su naredni tipovi iteratora

  `iterator`
  `const_iterator`
  `reverse_iterator`
  `const_reverse_iterator`

Neke metode klase vector, list i deque

- Svaka od ovih kolekcija sadrži i neke specifične metode koje ostale kolekcije ne sadrže

```
vector<int> niz;
for( int i=0; i<20; i++ )
    niz.push_back( i );

// I nacin
for( int i=0; i<niz.size(); i++ )
    cout << niz[i] << ' ';
cout << endl;
//0 1 2 3 4 ... 19

// II nacin
vector<int>::iterator
    b = niz.begin(),
    e = niz.end();
for( vector<int>::iterator i=b; i!=e; i++ )
    cout << (*i) << ' ';
cout << endl;
//0 1 2 3 4 ... 19
```

```
#include <vector>
...
    vector<int> niz;     // podrazumevani konstruktor pravi prazan niz
    cout << niz.size() << endl;     // 0
    vector<int> niz2(20);
    cout << niz2.size() << endl;     // 20
    for( int i=0; i<20; i++ )
        niz2[i] = i;    // elementima vektora pristupamo pomocu operatora[]
    niz.resize( 50 );    // promenu velicine niza mozemo izvoditi eksplictno...
    niz2.resize( 10 );
    cout << niz.size() << endl;     // 50
    niz.push_back( 25 );    // ...ili u koracima za po jedan
    cout << niz.size() << ' ' << niz.back() << endl;     //51 25
    niz.pop_back();
    cout << niz.size() << endl;    // 50
    // metodi za rad sa alociranim prostorom
    cout << niz.capacity() << endl;     //100
    niz.reserve(200);
    cout << niz.capacity() << endl;      //200
    cout << niz.size() << endl;    //50
...
```

```
#include <vector>
...
    vector<int> niz;
    for( int i=0; i<20; i++ )
        niz.push_back(i);
    for( int i=0; i<niz.size(); i++ )
        cout << niz[i] << ' ';
    cout << endl;      // 0 1 ... 19

    niz.insert( niz.begin() + 5, 100 );
    for( int i=0; i<niz.size(); i++ )
        cout << niz[i] << ' ';
    cout << endl;    //0 1 2 3 4 100 5 6 ... 19

    niz.erase( niz.begin() + 5 );
    for( int i=0; i<niz.size(); i++ )
        cout << niz[i] << ' ';
    cout << endl;      //0 1 2 ... 19
```

```
...
    niz.insert( niz.begin()+5, niz.begin(), niz.begin()+2);
    for( int i=0; i<niz.size(); i++ )
        cout << niz[i] << ' ';
    cout << endl;     //0 1 2 3 4 0 1 5 6 ... 19

    niz.erase( niz.begin()+5, niz.begin()+12 );
    for( int i=0; i<niz.size(); i++ )
        cout << niz[i] << ' ';
    cout << endl;     //0 1 2 3 4 10 11 12 ... 19
```

Primer

```
list<int> lista;
for( int i=0; i<20; i++ )
    lista.push_back(i);
for( list<int>::iterator i=lista.begin(); i!=lista.end(); i++ )
    cout << (*i) << ' ';
cout << endl;    //0 1 2 ... 19

list<int>::iterator i = lista.begin();
// Da bi pristupili nekom elementu liste
// neophodno je da pristupimo i svim prethodnim elementima.
i++; i++; i++; i++; i++;
lista.insert( i, 100 );
for( list<int>::iterator i=lista.begin(); i!=lista.end(); i++ )
    cout << (*i) << ' ';
cout << endl;    // 0 1 2 3 4 100 5 6 ... 19
```

Primer

```
lista.erase( i ); // Brise element na koji pokazuje i.
for( list<int>::iterator i=lista.begin(); i!=lista.end(); i++ )
    cout << (*i) << ' ';
cout << endl;     // 0 1 2 3 4 100 6 7 ... 19

for( int i=0; i<10; i++ )
    lista.push_front(i);
for( list<int>::iterator i=lista.begin(); i!=lista.end(); i++ )
    cout << (*i) << ' ';
cout << endl;    // 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 100 6 7 8 ... 19
```

Šablonska funkcija — prolazak kroz elemente kolekcije

```
template<typename kolekcija>
void ispisiSveElemente( const kolekcija& k )
{
    typename kolekcija::const_iterator
        i = k.begin(),
        e = k.end();
    for( ; i!=e; i++ )
        cout << (*i) << ' ';
    cout << endl;
}
...
    vector<int> niz;
    for( int i=0; i<20; i++ )
        niz.push_back( i );
    ispisiSveElemente( niz );
    // 0 1 ... 19
```

Šablonska funkcija — prolazak kroz elemente kolekcije

```
template<typename kolekcija>
void ispisiSveElemente( const kolekcija& k )
{
    typename kolekcija::const_iterator
        i = k.begin(),
        e = k.end();
    for( ; i!=e; i++ )
        cout << (*i) << ' ';
    cout << endl;
}
...
    list<float> lista;
    for( float x=0; x<50; x+=1.35 )
        lista.push_back( x );
    ispisiSveElemente( lista );
    // 0 1.35 2.7 ... 49.95
```

Šablonska funkcija — prolazak kroz elemente kolekcije

```
template<typename kolekcija>
void ispisiSveElemente( const kolekcija& k )
{
    typename kolekcija::const_iterator
        i = k.begin(),
        e = k.end();
    for( ; i!=e; i++ )
        cout << (*i) << ' ';
    cout << endl;
}
...
    set<char> znaci;     char* s="neki znaci";
    for( char* p=s; *p; p++ )
        znaci.insert( *p );
    ispisiSveElemente( znaci );
    //a c e i k n z
```

Primer

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

Primer

```
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;
}
```

Primer

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

Primer

```
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;
}
```

Primer

```
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;
}
```

Primer

```
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;
}
```

Primer

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

Primer

```
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 )
{...}
```

Šablon STL-a `count_if`

Funkcionali

- Funkcionalima se nazivaju klase čije instance mogu da se primene kao funkcije
- U nekim funkcijama ili metodama klasa standardne biblioteke potrebno je da se kao argument navede funkcija koja radi neki posao ili izračunava neku vrednost
- Posebno se često upotrebljavaju funkcije koje proveravaju da li je ispunjen neki uslov
- Funkcije su nepromenljivi objekti i to može nekada da bude problem

Primer

- Želimo da prebrojima za koliko elemenata neke kolekcije je zadovoljen neki uslov. Možemo koristiti šablon `count_if`

```
vector<int> v;
...
int n = 0; //rezultat prebrojavanja
count_if(v.begin(), v.end(), uslov, n);
```

- Ako unapred znamo koji uslov treba proveriti, onda možemo definisati funkciju uslov

```
bool veciOd5( double n ){
return n > 5;
}
...
count_if(v.begin(), v.end(), veciOd5, n);
```

Primer

- Primer možemo uopštiti uvođenjem šablona

```
template<int osnovaPoredjenja>
bool veciOd( int n ){
  return n > osnovaPoredjenja;
}
...
count_if(v.begin(), v.end(), veciOd<5>, n);
count_if(v.begin(), v.end(), veciOd<15>, n);
```

- Međutim, šabloni se statički instanciraju, pa ne možemo da uradimo ovako nešto

```
...
count_if(v.begin(), v.end(), veciOd<i>, n);
```

Funkcionali

- Umesto "dinamički instanciranih šablona" koriste se funkcionali
- Funkcional je objekat koji ume da primeni operator aplikacije () tj koji ume da nešto izračuna za date argumente

```
class VeciOd {
public:
  VeciOd(int n) : _N(n) {}
  bool operator() (int x) const
    { return x > _N; }
private:
  const int _N;
};
```

Primer

```
vector<int> v;
...
int k;
cin >> k;
VeciOd veciOdK(k);
int n;
count_if(v.begin(), v.end(), veciOdK, n);
```

ili

```
count_if(v.begin(), v.end(), veciOd(k), n);
```

Funkcionali

- Funkcije standardne biblioteke najčešće upotrebljavaju funkcionale sledećih oblika
  - Unarna funkcija
  - Binarna funkcija
  - Unarni predikat
  - Upoređivač (proverava da li je prvi argument strogo manji od drugog u datom kontekstu)
  - Transformator — unarni funkcional koji na određen način menja argument prenesen po referenci

Apstraktne kolekcije

- STL definiše tri apstraktne kolekcije: stek, red i red sa prioritetom
- Ove kolekcije ne omogućavaju neposrednu primenu iteratora
- Stek — elementi se dodaju, uzimaju i mogu koristiti samo sa vrha
- Red — elementi se dodaju na kraj reda, uzimaju sa početka, a mogu se koristiti i sa početka i sa kraja
- Red sa prioritetom — elementi se dodaju u red na mesto koje im odgovara na osnovu date funkcije prioriteta, koristi se i briše samo element sa najvećim prioritetom (tj sa početka reda)

Stek

- Za implementaciju se podrazumevano koristi dek

```
template< typename T, typename Container = deque<T> >
class stack;
```

- Konstruktor praznog ili inicijalno popunjenog steka

```
stack (const Container & = Container() )
```

- Metodi push, pop, top, empty, size

Red

- Za implementaciju se podrazumevano koristi dek

```
template< typename T, typename Container = deque<T> >
class queue;
```

- Konstruktor praznog ili inicijalno popunjenog stka

```
queue (const Container & = Container() )
```

- Metodi push, pop, front, back, empty, size

Red sa prioritetom

- Za implementaciju se podrazumevano koristi vector

```
template< class T, class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue;
```

  Parametar Compare predstavlja binarni funkcionalni tip koji proverava da li je prvi argument manji od drugog (definiše prioritet). Ukoliko se ne navede eksplicitno, upotrebljava se operator poređenja $<$.
- Konstruktor praznog ili inicijalno popunjenog reda sa prioritetom
- Metodi push, pop, top, empty, size

Uređene kolekcije

- Skupovi `set` i katalozi `map` predstavljaju kolekcije čiji su elementi međusobno uređeni
- Omogućavaju brz pristup elementima, odnosno brzo proveravanje da li traženi element pripada kolekciji ili ne
- Upotreba iteratora je u skladu sa uređenjem
- Sličan skup metoda
- Postoje i varijante koje dopuštaju ponavljanje elemenata `multiset` i `multimap`

Set i map

```
template< class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
class set;

template< class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator< pair<const Key, T> >
class map;
typedef pair<const Key, T> value_type;
```

Set i map

- Parametar `Key` određuje tip elemenata ključa, za tip `Key` nema ograničenja
- Za katalog za `T` mora da postoji javan podrazumevani konstruktr `T::T()`.
- U katalogu je moguće menjati vrednost, jednom postavljen ključ nije moguće menjati
- `Compare` određuje način poređenja objekata klase `Key`, podrazumeva se operator $<$,
- `Allocator` određuje način alociranja novih elemenata skupa ili kataloga, podrazumea se `new`.

Set i map — metode

- Konstruktor bez argumenata `set()` i `map()`
- Početni i završni iteratori:
  `iterator begin()`, `iterator end()`,
  `reverse_iterator rbegin()`, `reverse_iterator rend()`.
- `size_type size() const` — veličina
- `size_type max_size() const` — maksimalna veličina
- `bool empty() const` — Da li je skup/katalog prazan
- `void clear() const` — Pražnjenje skupa/kataloga
- `size_type count(const Key& x) const`
  Provera da li element pripada skupu ili katalogu, vraća 1 ako pripada, 0 ako ne pripada

Set i map — metode

- Dodavanje elemenata

```
pair<iterator, bool> insert(const value_type& x)
iterator insert(terator position, const value_type& x)
template <class InputIterator>
  void insert(InputIterator first, InputIterator last)
```

- Prvi dodaje element, drugi sugeriše na koje mesto ubaciti element, treći dodaje sve elemente iz opsega odgovarajućih iteratora
- Ukoliko već postoji odgovarajući element, ne izvodi se njegovo dodavanje, u trećem slučaju dodaju se samo oni koji ne postoje u kolekciji
- Rezultat prvog metoda je par: iterator na dodati element ili na pronađen ukoliko je već postojao, i indikator dodavanja (true

Set i map — metode

Brisanje elemenenata skupa/kataloga

```
size_type erase(const key_value& x) //broj obrisanih 1/0
void erase (iterator position)
void erase (iterator first, iterator last)
```

Set i map — metode

- Metodi za traženje elemenata

```
iterator find(const key_value& x)
iterator lower_bound(const key_type& x) //prvi koji nije manji
iterator upper_bound(const key_type& x) //prvi koji jeste veci
```

- Ako pretraga ne uspe, vraca se `end()`

Map opertor []

- Operator `[]` za neposredno pristupanje podatku koji odgovara ključu
  `T& operator[](const Key& x)`
- Ukoliko ne postoji element sa datom vrednošću ključa, on se automatski dodaje i pridružuje mu se podrazumevana vrednost podatka
- Primena ovog operatora je uvek uspešna

Multiset i multimap

- Kod `multiset`-a i `multimap`-a metodi dodavanja uvek uspevaju
- Metod `count` vraća broj koji može biti veći od 1, u zavisnosti od toga koliko je elemenata pronađeno
- Metod `erase` briše sve vrednosti sa istim ključem, ne samo jednu vrednost, vraća broj obrisanih elemenata
- `find` može da nađe bilo koji element

Ostale kolekcije

- Podržane kolekcije se odlikuju visokim nivoom apstrakcije
- Ne postoji podrška za drveta, višedimenzione nizove i grafove, ali:
  - Binarno drvo se obično upotrebljava za implementaciju skupova (`set`) ili kao osnova za brzo pretraživanje (`map`)
  - Višedimenzioni nizovi se mogu implementirati preko šablona klase `vector` jer `vector` može imati kao elemente bilo koji drugi tip, pa i sam `vector`
  - Grafovi se efikasno mogu implementirati primenom kataloga

Algoritmi

- Algoritimi — šabloni funkcija koji implementiraju pretraživanje, uređivanje, inicijalizovanje, transformaciju ...
- Dva važna mehanizma: apstrahovanje tipova podataka se postiže primenom šablona, a apstrahovanje tipova kolekcija se postiže primenom iteratora
- Ista implementacija algoritma se može primenjivati na sve tipove podataka i kolekcija
- `http://www.cplusplus.com/reference/algorithm/`

Literatura

- OOP — C++ kroz primere. Saša Malkov. Matematički fakultet, 2007.
- C++ izvornik. Sttanley B. Lippman, Josee Lajoie. CET
- Parametarski polimorfizam. Saša Malkov.