Objektno orijentisano programiranje 1

Generički mehanizam

Potreba za generičkim tipovima

- Isti algoritam je često potrebno izvršavati nad različitim tipovima podataka
- Primeri:
  - maksimum od dva podatka (cela, realna, znakovna, kompleksna,...)
  - kružni bafer ili stek
- Bez generičkog mehanizma trebalo bi pisati po jednu funkciju ili klasu za svaki od tipova podataka:

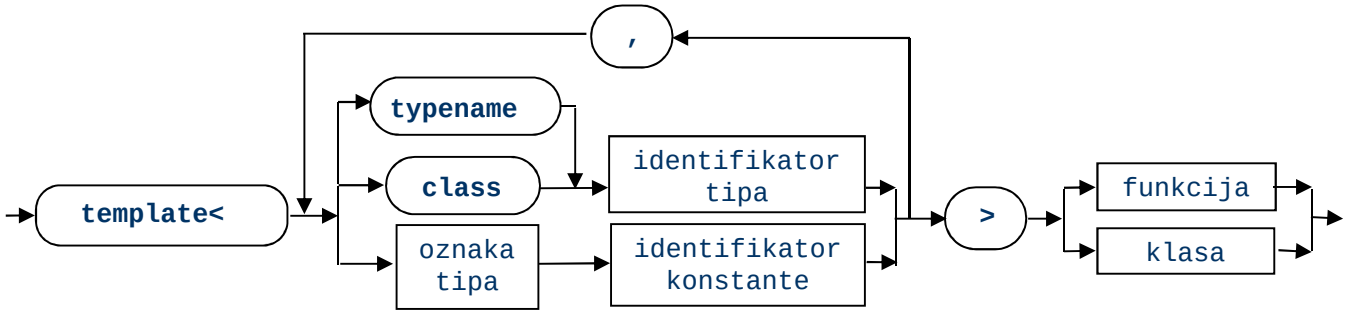
```
char      max(char i,char j)   {return i>j?i:j;}
int       max(int i,int j}     {return i>j?i:j;}
float     max(float i,float j}{return i>j?i:j;}
```

Šabloni u jeziku C++

- C++ omogućava definisanje šablona (*template*) za funkcije, gde je tip podatka argument šablona
- Na osnovu šablona se mogu automatski generisati konkretne funkcije za konkretan tip podataka
- Šabloni mogu da se prave i za klase
- Funkcije i klase opisane šablonom nazivaju se generičke
- Mehanizam generika je statički
  - konkretizacije šablona se prave tekstualnom zamenom u vreme prevođenja
- Odvojeno prevođenje šablona nema smisla
  - šabloni se smeštaju u *.h datoteke i uključuju tamo gde se koriste
- Mana šablona (pošto su u *.h datotekama):
  - korsnik vidi celu implementaciju algoritama, a ne samo interfejs

Definisanje šablona

- Sintaksni dijagram definicije generičke funkcije i klase:



- *funkcija* – deklaracija (prototip) ili definicija funkcije
- *klasa* – deklaracija ili definicija klase
- `typename` i `class` – formalno nema razlike
  - označavaju formalne (simboličke) proizvoljne tipove
  - `class` je starijeg datuma, neprirodno za označavanje primitivnih tipova
  - neformalno: `class` za klasne tipove, a `typename` za sve

Formalni parametri šablona

- Formalni parametri šablona između zagrada < i >
  - mogu predstavljati tipove ili konstante
- *identifikator tipa* – formalni tip
  - može da se koristi unutar šablona na svim mestima gde se očekuje identifikator tipa podatka
- *identifikator konstante* – simbolička konstanta
  - može da se koristi unutar šablona na mestima konstanti
- *oznaka tipa* – određuje tip simboličke konstate i može biti:
  - oznaka celobrojnog tipa (uključujući nabrajanja)
    - stvarni argument mora biti konstantni izraz
  - nepromenljiv pokazivač ili referenca
    - stvarni argument mora da upućuje na objekat ili funkciju sa spoljašnjim povezivanjem

Primeri šablona funkcija

```
//sablon funkcije:
   template <typename T> T max(T a, T b){return a>b?a:b;}
//sablon deklaracije (prototipa) funkcije:
   template <typename T> void sort(T a[], int n);
//sablon definicije funkcije:
   template <typename T> void sort(T a[], int n){/*...*/}

//odlozeno navodjenje tipa sablonske funkcije
   template <typename T, typename U>
   auto f(T t, U u)->decltype(t+u){return t+u;}
```

Primer šablona klase

```
//sablon klase
   template <class T, int k> class Vekt{
     int duz; T niz[k];
   public:
     Vekt();
     T& operator[](int i) const{return niz[i];}
   };
   template <class T, int k> Vekt<T,k>::Vekt(){
     duz=k; for(int i=0; i<k; niz[i++]=T());
   }
```

- U definiciji metode generičke klase, uz ime klase u operaciji `::` mora da stoje i argumenti
  - razlog: sam identifikator generičke klase ne definiše jednoznačno klasu

Generisanje funkcija (1)

- Konkretne funkcije se mogu generisati iz šablona:
  - automatski
  - na zahtev
- Automatski, na mestu poziva:
  - eksplicitnim navođenjem stvarnih argumenata šablona
    - iza imena funkcije se navedu stvarni argumenti šablona između <...>
    - tad su moguće i konverzije stvarnih argumenata funkcije u formalne
    - nepohodno kada argument šablona nije i argument funkcije
      - npr. argument šablona može biti tip rezultata funkcije
  - implicitnim zadavanjem stvarnih argumenata šablona
    - bez navođenja stvarnih argumenata šablona između <...>
    - na osnovu stvarnih argumenata funkcije se određuju argumenti šablona
    - nisu moguće konverzije argumenta funkcije
  - moguće je i navođenje samo nekoliko prvih argumenata šablona

Generisanje funkcija (2)

- Automatsko implicitno generisanje funkcije će biti sprečeno:
  - ako se prethodno pojavi definicija odgovarajuće obične funkcije
    - ako argumenti poziva funkcije bez konverzije odgovaraju parametrima
- Forsirano generisanje funkcije iz šablona
  u slučaju postojanja odgovarajuće definicije obične funkcije
  - eksplicitnim navođenjem stvarnih argumenata šablona
  - funkcija se generiše iz šablona, ne poziva se postojeća definicija
- Generisanje na zahtev se postiže:
  - navođenjem deklaracije sa tipovima stvarnih argumenata šablona
  - **`template`** *`tip ime <stvarni_arg>(lista_tipova);`*
  - primer: **`template int`** `max<`**`int`**`>(`**`int`** `a,` **`int`** `b);`

Primer generisanja funkcija

```
const char *max(const char *cp1,const char *cp2){ // def. obicne f-je
  return strcmp(cp1,cp2)>=0?cp1:cp2;    // da se spreci poredjenje pok.
}

template float max <float> (float,float); // zahtev za generisanje

void main() {
  int i1=1, i2=2; float f=0.5;
  char c1='a', c2='b'; const char *a="alfa", *b="beta";
  int i=max(i1,i2);              // generise se int max(int,int)
              // ili: int i=max<int>(i1,i2);
  char c=max(c1,c2);             // generise se char max(char,char)
  int g=max(i1,c1);              // ! GRESKA
              // moze: int g=max<int>(i1,c1); // konverzija char u int
  float g=max<float>(i1,f);      // generise se float max(float,float)
  const char *v=max(a,b);        // poziva se obicna f'ja, poredi niske
  const char *w=max<const char*>(a,b);//poziva se gen. f-ja, poredi pok.
}
```

Generisanje klasa

- Konkretna klasa se generiše kada se prvi put naiđe na definiciju objekta u kojoj se koristi identifikator te klase
- Pri generisanju klase se generišu i sve funkcije članice
- Oznaka tipa generisane klase treba da sadrži:
  - identifikator generičke klase i
  - listu stvarnih argumenata za generičke tipove i konstante unutar <> iza identifikatora klase
- Stvarni argument može biti:
  - oznaka tipa – zamenjuje formalni argument koji je identifikator tipa
    - oznaka tipa može biti standardni tip, ime klase ili izvedeni tip (npr. pokazivač)
  - konstantni izraz – zamenjuje formalni argument koji je identifikator konstante
    - ako izraz sadrži operator > ovaj se mora pisati u zagradama (>)
- I klasa se može generisati na zahtev: `template class` *`ime`*`<`*`arg`*`>;`
- Generisana klasa se može imenovati (`typedef` ili `using`)

Primeri generisanja klasa

- Automatsko generisanje

```
Vekt<int,10> niz1;
Vekt<double,20> niz2;
class A{int i;};
Vekt<A,5> a;
```

- Generisanje na zahtev

```
template class Vekt<char,10>;
```

- Generisanje na zahtev uz imenovanje tipa

```
typedef Vekt<char*,100> VektorStr;
using VektorRealnih=Vekt<float, 1000>;
```

Podrazumevane vrednosti

- Argumenti šablona mogu da imaju podrazumevane vrednosti (PV)
- Nekoliko poslednjih argumenata šablonske klase može da ima PV
  - slično kao kod običnih funkcija
- Šablonske funkcije ne mogu imati PV argumenata šablona
- Dovoljno je navesti PV samo u deklaraciji/definiciji šablonske klase
  - nije potrebno ponovo u definicijama metoda klase
- Ako je (formalni) parametar šablona tip:
  - PV može da bude: primitivan tip, ne-generička klasa, generička klasa sa stvarnim argumentima šablona
- Ako je (formalni) parametar šablona konstanta:
  - PV može da bude konstantni celobrojni izraz

Primer podrazumevanih vrednosti

```
template<typename T=int, int k=10> class Niz{/*…*/}
Niz<char,55> n1; Niz<float> n2; Niz<> n3;
```

- Raniji formalni argumenti mogu da se koriste za određivanje početnih vrednosti kasnijih argumenata

```
template<typename T, int k,
   typename U=Niz<short>, typename V=Niz<T,k> >
   class A{/*…*/}
```

Funkcijske klase kao parametri

- Funkcije ne mogu biti parametri šablona
- Korisno je da se u šablonima može varirati i obrada
- Rešenje: funkcijska klasa – ima `operator()`
  - kao klasa `F` može biti parametar šablona
  - preko objekta te klase `F f` se sa `f();` poziva funkcija `operator()`
- Pri generisanju konkretne funkcije
  - stvarni argument funkcije je objekat funkcijske klase
  - može biti i lambda funkcija (anonimni funkcijski objekat)
- Pri generisanju konkretne klase
  - stvarni argument šablona je funkcijska klasa
  - ne može biti lambda funkcija

Primeri funkcijskih parametara

- Šablon funkcije za tabeliranje proizvoljne funkcije:

```
template<typename T, class F>
  void tabela(F f, T xmin, T xmax, T dx){
    for (T x=xmin; x<=xmax; x+=dx)
      cout<<x<<'\t'<<f(x)<<endl;
  }
```

- Šablon klase za tabeliranje proizvoljne funkcije:

```
template<typename T, class F> class Tabela {
  F f;
public:
  void tabela(T xmin, T xmax, T dx){
    for (T x=xmin; x<=xmax; x+=dx)
      cout<<x<<'\t'<<f(x)<<endl;
  }
};
```

Primeri funkcijskih argumenata

- Generička funkcijska klasa za računanje prigušenih oscilacija
  - `T` može biti `float`, `double`, `long double`, ali čak i klasa kompleksnih brojeva

```
template<typename T> class Oscil {public:
  T operator()(const T& x) const
    { return exp(-0.1*x)*sin(x); }
};
```

- Generisanje funkcije i klase za tabeliranje prigušenih oscilacija

```
tabela(Oscil<double>(), 0., 6.28, 0.314); cout<<endl;

Tabela<double, Oscil<double> > t;
t.tabela(0., 6.28, 0.314);
```

Inicijalizatorske liste

- Standardna generička klasa za opis inicijalizatorske liste

```
template<typename E> class initializer_list{ public:
  initializer_list() noexcept;
  size_t size() const noexcept;
  const E* begin() const noexcept;
  const E* end() const noexcept;
}
```

- Konstruktor klase za neprazne liste je privatan
  - poziva ga samo prevodilac za stvaranje ini. liste na osnovu navedenih podataka
  - javnim konstruktorom se stvara samo prazna lista
  - operatorom dodele može da se praznoj ini. listi dodeli druga ini. lista
- Metoda `size()` vraća broj elemenata liste
- Metoda `begin()` vraća pokazivač na nepromenljiv prvi element liste
- Metoda `end()` vraća pokazivač iza poslednjeg elementa liste

Primer inicijalizatorske liste

```
initializer_list<double> ilist {1, 2.2, 3.3F, '4', 5};
int n = ilist.size(); cout<<n<<endl; // 5
const double* p=ilist.begin();
for(int i=0; i<n; i++) cout<<p[i]<<' '; cout<<endl;
for(const double* p=ilist.begin(); p!=ilist.end(); p++)
  cout<<*p<<' ';
cout<<endl;
for(double x: ilist) cout<<x<<' '; cout<<endl;
```

- Primeri ini. liste kao parametra/argumenta i rezultata funkcije

```
initializer_list<float> g(int x,
            initializer_list<double> y, char c)
  { return {x, 2.2F, c, 5.5F, 6L, 7LL };
initializer_list<float> a=g(1,{2,3,4,5},'a');
auto b=g(1, {2,3,4,5},'a');
```

Inicijalizacija objekata inic. listom

- Objekat može da se inicijalizuje inic. listom
  - potrebno je da klasa ima konstruktor sa parametrom inic. liste
  - ostali parametri konstruktora moraju imati podrazumevane vrednosti
- Argumenti konstruktora su uobičajeno u zagradama ( )
  - mogu da se navedu i u vitičastim zagradama {*niz izraza*}
  - izrazi mogu da budu različitih tipova
- Ako u klasi postoji konstruktor čiji je parametar inic. lista
  - obavezno se poziva taj konstruktor kada se argumenti navedu u {}
  - izuzetno – u slučaju praznog niza izraza u zagradama {}, kada se poziva podrazumevani konstruktor

Primer inicijalizacije objekta

```
class A {
  int* niz; int n;
public:
  A():niz(nullptr), n(0){}
  A(int x, int y):niz(new int[2]{x,y}),n(2){}
  A(initializer_list<int> ilist){
    niz = new int [n=ilist.size()];
    int i=0; for(int x:ilist)niz[i++]=x;
  }
}
A a1{1,2,3,4,5};    // A(initializer_list<int>)
A a2;               // A()
A a3{};             // A()
A a4{1};            // A(initializer_list<int>)
A a5{1, 2};         // A(initializer_list<int>)
A a6(1, 2);         // A(int, int)
```

Specijalizacija

- Ponekad je za neke specifične vrednosti argumenata šablona potrebna specifična definicija šablonske funkcije ili klase
- Specijalizacija šablona (f-je, klase):
  - definisanje posebnog šablona za neke posebne kombinacije argumenata prethodno definisanog (opšteg) šablona
- Specijalizacija generičke klase:
  - deklaracija:
    `template<`*`parametri`*`>class GSklasa<`*`argumenti`*`>;`
  - definicija:
    `template<`*`parametri`*`>class GSklasa<`*`argumenti`*`>{/*…*/}`
  - `GSklasa` – generička specijalizovana klasa
  - *`parametri`* – parametri specijalizovanog šablona
    - svi ili samo neki parametri opšteg šablona
    - ne mogu imati podrazumevane vrednosti
  - *`argumenti`* – određuju varijantu opšteg šablona
    - fiksiraju neke (ili čak sve) vrednosti parametara opšteg šablona za datu specijalizaciju

Vrste i uslovi specijalizacije

- Specijalizacija može da bude:
  - delimična: ako specijalizovani šablon ima barem jedan parametar
    - tada može da se generiše više klasa iz specijalizovanog šablona
  - potpuna: specijalizovani šablon nema ni jedan parametar
    - tada na osnovu šablona može da se generiše samo jedna klasa
- Specijalizacija je moguća:
  - ako je prethodno navedena barem deklaracija opšte generičke klase
- Specijalizacija određenim parametrima je moguća:
  - samo ako još nije generisana ni jedna klasa na osnovu opšteg šablona sa istim parametrima

Primer specijalizacije

```
template<typename T, int k> class Vekt; //opsta gen. klasa
template<typename T, int k> class Vekt<T*,k>;       // spec 1
template<typename T> class Vekt<T,10>;              // spec 2
template<int k> class Vekt<int,k>;                  // spec 3
template<> class Vekt<void*,10>;                    // spec 4
Vekt<int*,15> n1;             // koristi se 1
Vekt<float,10> n2;            // koristi se 2
Vekt<int,20> n3;              // koristi se 3
Vekt<void*,10> n4;            // koristi se 4
```

- Pri generisanju se navodi onoliko elemenata koliko ima opšti šablon
- Prevodilac bira šablon:
  - najviše specijalizovani (sa najmanje parametara), pa manje specijalizovani
  - opšti (ako ni jedan specijalizovani ne odgovara)
  - ako postoji više podjednako specijalizovanih šablona koji odgovaraju – greška
    na primer: `Vekt<int,10> n5;`

Specijalizacija generičkih f-ja

- Za generičke funkcije – moguća je samo potpuna specijalizacija
  - deklaracija:
    ```
    template<>tip GSfunkcija<argumenti>(...);
    ```
  - definicija:
    ```
    template<>tip GSfunkcija<argumenti>(...){/*…*/}
    ```
- Argumenti (uključujući i <>) mogu da se izostave
  - ako se na osnovu tipova argumenata funkcije mogu odrediti vrednosti (tipovi) argumenata šablona
- Primer: umesto posebne definicije, specijalizacija, da se izbegne poređenje pokaz.

```
template<>char* max<char*>(char*a,char*b);// ili:
template<>char* max (char*a,char*b);
char* m=max<char*>("alfa", "beta"); // ili:
char* n=max("alfa", "beta");
```

  - poziva definisanu (negeneričku) funkciju max ako takva postoji

Generičke metode

- Metode klase mogu da budu generičke
  - bez obzira na to da li je klasa generička ili ne
- Generičke metode mogu da se definišu unutar klase ili izvan
- Generičke metode ne mogu da budu virtuelne
- Destruktor ne može da bude generički
- Konstruktori mogu da budu generički
- Kod poziva generičkih metoda:
  - ako su u izrazima posle operatora `::` `.` `i` `->` `i`
  - ako se iza imena metode navode argumenti šablona
  - onda ispred imena metode treba navesti reč `template` (neki prevodioci ne traže)

Primeri generičkih metoda (1)

- Generički konstruktor i generička metoda obične klase

```
class A{
  double x;
public:
  template<typename T> A(const T&t):x(t){}
  template<typename T> void m(const T& t);
};
template <typename T> void A::m(const T&t){x=t;}
void f1(){
  A a1(5);        // A::A(const int&)
  A a2(1.2);      // A::A(const double&)
  A a3('q');      // A::A(const char&)
  a1.template m<int>(1.2);         // A::m(const int&)
  a1.template m<char>(3.4);        // A::m(const char&)
  a1.m(3.4);                       // A::m(const double&)
}
```

Primeri generičkih metoda (2)

- Generička metoda generičke klase

```
template <typename T> class B
  T t;
public:
  template<typename U> void m(const U&);
};
template <typename T> template<typename U> void B<T>::m(const U& u){t=u;}
void f2(){
  B<int> b1;
  B<float> b2;
  b1.m(55);                          // B<int>::m(const int&)
  b2.m(55);                          // B<float>::m(const int&)
  b2.template m<char>(55);           // B<float>::m(const char&)
}
```

Unutrašnje generičke klase

- Unutrašnje klase mogu da budu generičke
  - njena spoljna klasa može da bude obična ili generička
  - u slučaju generičke spoljne, odnosno unutrašnje klase, oznaka klase mora da sadrži i argumente šablona
- Primer:

```
template <typename T> class C{
public: template<typename U> class D;
};
template<typename T> template <typename U> class C<T>::D{
public: void m (const U&);
};
template <typename T> template <typename U> void C<T>::D<U>::m(const U& u){
}
void f3(){
  C<int> c; C<double>::D<char> d; d.m(true); // C<double>::D<char>::m(char)
}                                            // true se konvertuje u char
```

Paketi parametara

- U jeziku C – funkcije sa promenljivim brojem parametara
  - npr. scanf i printf: prvi parametar daje uputstvo za tipove ostalih
  - nebezbedno – prevodilac ne proverava, ne otkriva se ni u vreme izvršenja
- Paketi parametara šablonskih funkcija (i klasa) – rešenje problema
- Parametar šablona koji može da prihvati proizvoljan broj argumenata
- Sintaksa:
  - `class ...`*`paket`*, odnosno `typename ...`*`paket`*, odnosno *`tip`* `...`*`paket`*
  - *`paket`* – identifikator paketa, formalno ime serije tipova/konstanti
- Primer:

```
template <typename …T> class A {}; // paket tipova
template <int …T> class B{};   // paket konstanti
```

- Paket parametara može biti samo poslednji parametar šablona
- Ne može da ima podrazumevanu vrednost

```
template<class S, class T=int, class …U> class C{};
```

Paketi parametara - generisanje

- Broj i tipovi parametara u paketu tipova se određuju pri generisanju
  - na osnovu broja i tipova argumenata
- Primer:

```
A<> a1;      // paket: prazan
A<int> a2;   // paket: int
A<double, int, char> a3;   // paket: double, int, char
B<> b1;      // paket: prazan
B<5> b2;     // paket: 5
B<5,'A',1>   // paket: 5, 65, 1
C<> c1;      // ! GRESKA, ne zna se tip S
C<int> c2;   // S=int, T=int, U={}
C<int, char, int, char, bool> c3;
             // S=int, T=char, U={int, char, bool}
```

Raspakivanje paketa parametara

- Sintaksa: *`paket`* …
- Dobije se serija tipova/podataka koji se dalje koriste kao argumenti
- Prvih nekoliko (mogu i svi) se koriste pojedinačno
  - ostali se ponovo upakuju u paket
- Primer:

```
template<typename S, typename …T> void f(S s, T… t){}
template<typename S, typename …T>
  void g(S s, T… t){ f(t…); }
void h(){ g(1,2.2,3L,4.4F); }
// g():S=int,s=1,T…={double,long,float),t=(2.2,3L,4.4F)
// f():S=double,s=2.2,T…={long,float),t=(3L,4.4F)
```

- Određivanje broja elemenata paketa - operatorom `sizeof...`

Upotreba u inicijalizatorskim listama

- Ako se inicijalizatorska lista koristi za inicijalizaciju niza ili u for naredbi
  - svi elementi paketa moraju biti istog tipa
- Primer:

```
template<typename …T> int f(T… t) {
  int d=sizeof…(t);
  int niz[]={t…};
  int s=0;
  for (int x:{t…})s+=x;
  cout<<“d=“<<d<<“, s=“<<s<<endl;
  return s;
}
int p = f(1,2,3); // d=3, s=6
int q = f(1, 2., 3); // ! GRESKA – raliciti tipovi
```

Upotreba u negeneričkoj metodi

- Paketi parametara mogu da se iskoriste samo kao argumenti
  - generičkih funkcija
  - metoda generičke klase
- Negenerička metoda može da se poziva samo sa tačnim brojem arg.
  - onoliko koliko tipova je navedeno pri generisanju klase
- Primer upotrebe u negeneričkoj metodi generičke klase:

```
template<typename …S> class D { public:
  void f(S… s){} // negenericka metoda generic ke klase
};
void p(){
  D<int,double> d;
  d.f(1);          // ! GRESKA – premalo argumenata
  d.f(1, 2.2);
  d.f(1, 2.2, 3); // ! GRESKA – previse argumenata
}
```

Rekurzivna obrada argumenata

- Opšti oblik rešenja obade promenljivog broja argumenata istog tipa:

```
template<typename …T> tip fun(T… t) { …
  for(auto e: {t…}) {// obrada e
  } …
} // nedostatak: svi argumenti moraju biti istog tipa
```

- Opštije rešenje:

```
tip f() {…} // potrebno za 0 argumenata
template<typename S, typename …T>
tip f(S prvi, T… ostali){
  … // obrada parametra prvi
  f(ostali…);
  … // obrada prametra prvi
}
```

Pisanje promenljivog broja podataka

```
#include <iostream>
 using namespace std;
 void pisi(){}
 template<typename S, typename …T> void pisi(S s, T… t){
   cout<<s;
   pisi(t…);
 }
 struct X {int a,b;);
 inline ostream& operator<<(ostream& it, const X& x)
   {return it<<‘(’<<x.a<<‘,’<<x.b<<‘)’;}
 X x={1,2};
 int main(){ pisi(“i=”,5,“ x=”,x,“ r=”, 1.234, ‘\n’); }
 // i=5 x=(1,2) r=1.234
```