Objektno orijentisano programiranje 1

Generički mehanizam

Potreba za generičkim tipovima

- Često je potrebno da se isti algoritam izvršava nad različitim tipovima podataka
- Primeri:
  - maksimum od dva podatka (cela, realna, znakovna, kompleksna,...)
  - kružni bafer ili stek
- Bez generičkog mehanizma trebalo bi da se piše po jedna funkcija (ili klasa) za svaki od tipova podataka:

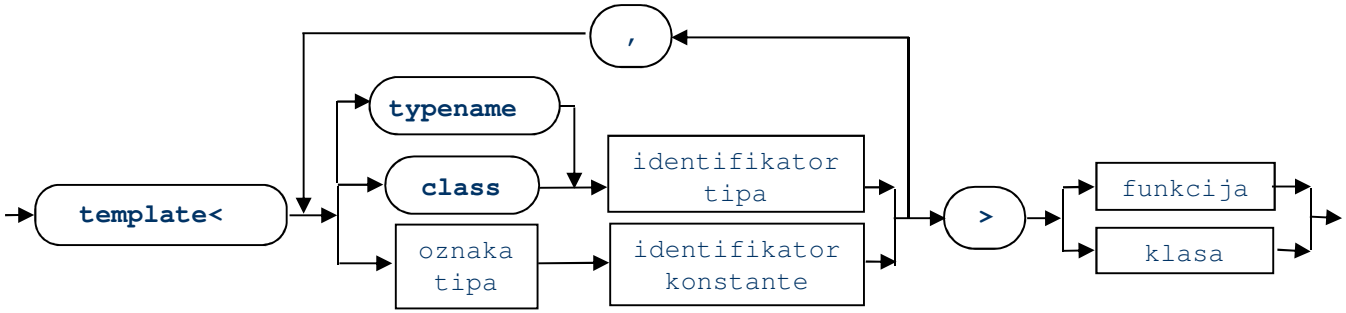
```
char      maximum(char i,char j)   {return i>j?i:j;}
int       maximum(int i,int j)     {return i>j?i:j;}
float     maximum(float i,float j){return i>j?i:j;}
```

Šabloni u jeziku C++

- C++ omogućava definisanje šablona (*template*) za funkcije, gde je tip podatka argument šablona
- Na osnovu šablona mogu da se automatski generišu 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 ugovor

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 da predstavljaju 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 da bude:
  - oznaka celobrojnog tipa (uključujući nabrajanja)
    - stvarni argument mora da bude konstantan izraz
  - pokazivač ili referenca

Primeri šablona funkcija

```
//sablon funkcije:
   template <typename T> T maximum(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();
     const 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 fuunkcija generičke klase, uz ime klase u operaciji `::` moraju da stoje i parametri šablona
  - razlog: sam identifikator generičke klase ne definiše jednoznačno klasu
  - zato je ime generičke klase u definiciji konstruktora (ili metoda): `template <class T, int k> Vekt<T,k>`

Generisanje funkcija (1)

- Konkretne funkcije mogu da se generišu iz šablona:
  - automatski
  - na zahtev
- Automatski, na mestu poziva:
  - eksplicitnim navođenjem stvarnih argumenata šablona
    - iza imena funkcije se navedu stvarni argumenti šablona između <...>
    - tada su moguće i konverzije stvarnih argumenata funkcije u parametre
    - neophodno kada argument šablona nije i argument funkcije
      - npr. argument šablona može da bude 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 da bude 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 funkcije sa argumentima šablona
  - **`template`** *`tip ime <arg_sablona>(lista_tipova);`*
  - primer: **`template int`** `maximum<`**`int`**`>(`**`int, int`**`);`

Primer generisanja funkcija

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

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


int main() {
  int i1=1, i2=2; float f=0.5;
  char c1='a', c2='b'; const char *a="alfa", *b="beta";
  int i=maximum(i1,i2);         // generise se int maximum(int,int)
     // ili: int i=maximum<int>(i1,i2);
  char c=maximum(c1,c2);        // generise se char maximum(char,char)
  int g=maximum(i1,c1);         // ! GRESKA
     // moze: int g=maximum<int>(i1,c1); // konverzija char u int
  float h=maximum<float>(i1,f);// generise se float maximum(float,float)
  const char *v=maximum(a,b);  // poziva se obicna f-ja, poredi niske
  const char *w=maximum<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 svi virtuelni metodi, a ostali pri pozivu
- 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 da bude:
  - oznaka tipa – zamenjuje parametar šablona koji je identifikator tipa
    - oznaka tipa može da bude standardni tip, ime klase ili izvedeni tip (npr. pokazivač)
  - konstantni izraz – zamenjuje parametar šablona koji je identifikator konstante
    - ako izraz sadrži operator `>` ovaj mora da se piše u zagradama `(>)`
- I klasa može da se generiše na zahtev: `template class` *`ime<arg>`*`;`
- Generisana klasa može da se imenuje (`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 šablona može da ima PV
  - slično kao kod običnih funkcija
- Šablonske funkcije mogu da imaju PV argumenata šablona
  - nebitne za parametre koji su tipovi argumenata f-je
  - bitne za parametre koji su tipovi rezultata f-je i za konstante
- Dovoljno je navesti PV samo u deklaraciji/definiciji šablonske klase
  - nije potrebno ponovo u definicijama metoda klase
- Ako je parametar šablona tip:
  - PV može da bude: primitivan tip, ne-generička klasa, generička klasa sa stvarnim argumentima šablona
- Ako je 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 parametri mogu da se koriste za određivanje početnih vrednosti kasnijih parametara

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

Funkcijske klase kao parametri

- Funkcije ne mogu da budu parametri šablona
- Korisno je da u šablonima može da se varira i obrada
- Rešenje: funkcijska klasa – ima `operator()`
  - kao klasa, `F` može da bude parametar šablona
  - preko objekta `F f` se pomoću `f()` poziva funkcija `operator()`
- Pri generisanju konkretne funkcije ili klase
  - stvarni argument šablona je funkcijska klasa
- Pri generisanju konkretne funkcije
  - stvarni argument funkcije je objekat funkcijske klase

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:
  Tabela(F fo){f=fo;}
  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 da bude `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;

Oscil<double> o;
Tabela<double, Oscil<double> > t(o);
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;
}
```

- Javnim podrazumevanim konstruktorom se stvara prazna lista
- Konstruktor klase za neprazne liste je privatan
  - poziva ga samo prevodilac za stvaranje ini. liste na osnovu navedenih podataka
- Metod `size()` vraća broj elemenata liste
- Metod `begin()` vraća pokazivač na nepromenljiv prvi element liste
- Metod `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 inicijalizatorskom listom
  - potrebno je da klasa ima konstruktor sa parametrom inic. liste
  - ostali parametri konstruktora moraju da imaju 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
- Ako u klasi ne postoji konstruktor čiji je parametar inic. lista
  - pri inicijalizaciji objetka se zagrade `{}` tretiraju kao obične zagrade

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, samo neki ili nijedan parametar opšteg šablona
    - ne mogu da imaju podrazumevane vrednosti argumenata
  - *`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 argumentima je moguća:
  - samo ako još nije generisana ni jedna klasa na osnovu opšteg šablona sa istim argumentima

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 argumenata koliko ima opšti šablon par.
- 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 na osnovu tipova argumenata funkcije
    mogu da se odrede vrednosti (tipovi) argumenata šablona
- Primer: umesto posebne definicije - specijalizacija, da se izbegne poređenje pokaz.

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

  - poziva definisanu (negeneričku) funkciju maximum ako takva postoji

Generički metodi

- Nevirtuelni metodi klase mogu da budu generički
  - bez obzira na to da li je klasa generička ili ne
- Generički metodi mogu da se definišu unutar klase ili izvan nje
- Virtuelni metodi ne mogu da budu generički
- Destruktor ne može da bude generički
- Konstruktori mogu da budu generički
- Kod poziva generičkih metoda:
  - iza imena metoda mogu da se navedu argumenti šablona u `<>`
  - nije potrebno eksplicitno navođenje argumenata šablona ako na osnovu argumenata metoda oni mogu da se odrede
  - ispred imena metoda može (ne mora) da se navede reč `template`

Primeri generičkih metoda (1)

- Generički konstruktor i generički metod 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;}
int main(){
  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&)
  return 0;
}
```

Primeri generičkih metoda (2)

- Generički metod 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;}
int main(){
  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&)
  return 0;
}
```

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 spoljne, odnosno unutrašnje klase mora da sadrži i parametre 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){
}
int main(){
  C<int> c; C<double>::D<char> d; d.m(65); // C<double>::D<char>::m(char)
  return 0;                                // 65 se konvertuje u char ('A')
}
```