Objektno orijentirano programiranje C++

Predavanje 02 - dedukcija parametara, moderni C++

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

04. ožujka 2023.

Primjer:

```
1 template <typename T>
2 void f(ParamType param);
3
4 f(expr);
```

Iz izraza `expr` prevodioc određuje tip predloška T i tip `ParamType`.

```
1 template <typename T>
2 void f(const T& param);
3
4 int x;
5 f(x); //T = int, ParamType = const int &
```

Detalji dedukcije parametra

Kada je `ParamType = T`, tip se određuje iz izraza `expr` tako da:

- Ako je `expr` referenca na tip, referenca se ignorira.
- Ako je izraz nakon zanemarivanja reference konstantan, konstantnost se ignorira (isto vrijedi za `volatile`).

```
1  template <typename T>
2  void f(T param);
3
4  int i = 2;
5  const int j = 4;
6  const int &k = j;
7
8  f(i); // T = int
9  f(j); // T = int
10 f(k); // T = int
```

Detalji dedukcije parametra

Kada je `ParamType` lijeva referenca, tip se određuje iz izraza `expr` tako da:

- Ako je `expr` referenca na tip, referenca se ignorira.
- Tip T se odredi iz izraza dobivenog ignoriranjem reference.

```
1  template <typename T>
2  T f(T& param){return param;}
3
4  int i = 4;
5  const int j = 6;
6  const int &k = j;
7
8  f(i); // T = int
9  f(j); // T = const int
10 f(k); // T = const int, ParamType = const int &
```

Detalji dedukcije parametra

```
1  template <typename T>
2  T f(const T& param){ return param; }
3
4  int i = 4;
5  const int j = 5;
6  const int &k = j;
7
8  f(i); //T = int, ParamType = const int &
9  f(j); //T = int, ParamType = const int &
10 f(k); //T = int, ParamType = const int &
```

Detalji dedukcije parametra

Kada je `ParamType = T&&` dobivamo različito ponašanje od desne reference. Ovaj slučaj se zove **univerzalna** referenca.

Parametri se deduciraju na sljedeći način:

- Ako `expr` ima lijevu vrijednost onda se T i T`&&` deduciraju kao lijeve reference.
- Ako `expr` ima desnu vrijednost, tip T se deducira na standardan način, tako da T`&&` bude desna referenca.

Detalji dedukcije parametra

```
1  template <typename T>
2  T f(T&& param){ return param; }
3
4  int i = 8;
5  const int j = 10;
6  const int &k = j;
7
8  f(i); //T = int&, ParamType = int &
9  f(j); //T = const int&, ParamType = const int &
10 f(k); //T = const int&, ParamType = const int &
11 f(18); //T = int, ParamType = int &&
12 f(k*k); //T = int, ParamType = int &&
```

Detalji dedukcije parametra

ParamType može biti složeniji i sadržavati više različitih parametara:

```
template <typename T, std::size_t dim>
T norm(std::array<T,dim> const & vec){
    T result = 0;
    for(std::size_t i = 0; i < dim; ++i)
        result += vec[i]*vec[i];

    return std::sqrt(result);
}
```

```
std::array<double,3> vv={1,5,3};
std::cout << norm(vv) << std::endl;
//T = double, dim = 3
//instancira norm(std::array<double,3> const &)
```

Adresa instance predloška

```
1 template <typename T>
2 void f(const T& t) { cout << t << endl; }
3
4 void (*pf)(const double&) = f;
5 //pf sadrzi adresu instance predloska funkcije f
6 //za T = double
```

Eksplicitno zadavanje parametara

Parametar predloška koji označava povratni tip funkcije **ne može se deducirati**. Njega navodimo **eksplicitno**.

```
1 template <typename T1, typename T2, typename T3>
2 T1 prod(T2 x, T3 y) { return x * y; }
3
4 double a = 2.3;
5 int b = 4;
6 double rez = prod(a,b); //greska
7 double rez = prod<double>(a,b);//OK!
```

Možemo zadati i preostale parametre:

```
1  double rez = prod<double>(a,b);
2  double rez1 = prod<double,double>(a,b);
3  double rez2 = prod<double,double,int>(a,b);
```

Preopterećenje funkcijskog predloška

Funkcijski predlošci se mogu preopteretiti:

```
1  template <typename T>
2  T Prod(T const& a, T const& b){
3      return a*b; }
4
5  // Preopterecena funkcija
6  int Prod(int const& a, int const& b){
7      return a*b; }
8
9  // Preopterecni predlozak
10 template <typename T>
11 T Prod(T const& a, T const& b, T const& c){
12     return  a*b*c; }
```

Nalaženje prave funkcije kod predefiniranih funkcija

1. Formira se skup funkcija kandidata koji sadrži:
   - Obične funkcije s istim imenom kao pozvana funkcija.
   - Sve instance svih predložaka funkcija koje imaju isto ime kao pozvana funkcija, za koje je postupak određivanja parametara doveo do slaganja argumenata u instanci i pozivu funkcije.
2. Određuje se koja obična funkcija se može pozvati s danim argumentima. Sve instance predložaka su dobri kandidati jer je algoritam određivanja parametara doveo do potpunog slaganja argumenata (do na konverzije).
3. Dobri kandidati se rangiraju prema konverzijama koje treba izvršiti da bi se osvario funkcijski poziv. Ako je obična funkcija jednako dobra prema konverzijama kao i parametrizirana funkcija, obična funkcija se smatra boljim kandidatom.
4. Od nekoliko najbolje rangiranih (jednako dobrih) kandidata parametriziranih funkcija, biramo onu s najspecijaliziranijim predloškom.

Nalaženje prave funkcije kod predefiniranih funkcija

Ako je na kraju izabrana samo jedna funkcija poštujući gore definirana pravila, **ona se poziva**. Ukoliko prateći pravila dobijemo više konačnih kandidata, **poziv je dvosmislen** i **program je neispravan**.

Ukoliko s $\succ$ označimo relaciju preferencije, tada:

- obične funkcije $\succ$ parametrizirane
- funkcije sa specijaliziranijim predlošcima $\succ$ funkcije s generalnijim predlošcima

Kod korištenja preopterećenih funkcija treba paziti na dvosmislene definicije i primjene.

Izabir prave funkcije - vježba

Koja funkcija će biti pozvana?

```
1  template <typename T>
2  T Max(T const& a, T const& b) {
3    return (a < b) ? b : a; }
4
5  int Max(int const& a, int const& b) {
6     return (a < b) ? b : a; }
7
8  template <typename T>
9  T Max(T const& a, T const& b, T const& c) {
10    return  Max(Max(a, b),c); }
11
12  Max(3,7,1); //treca verzija, T = int
13  Max(3.0,7.0); //prva verzija, T = double
14  Max('a','b'); //prva verzija
15  Max(3,7); //druga verzija
```

Izabir prave funkcije - vježba

Koja funkcija će biti pozvana?

```
1  Max<>(3,7) //prva verzija, T = int
2  Max<double>(3,7) //prva verzija, T = double
3  Max(3.0,7) //prva verzija, T = double
```

Ukoliko funkcijski poziv odgovara i običnoj funkciji i generičkoj funkciji, možemo definirati da se pozove generička funkcija koristeći $<>$ ispred liste parametara.

Kod poziva obične funkcije se vrše standardne konverzije dok su kod predloška dozvoljene samo trivijalne konverzije. Ukoliko želimo izvoditi standardne konverzije pri pozivu funkcije s predloškom, trebamo eksplicitno definirati parametar predloška.

Izabir prave funkcije - vježba

```
1  #include <iostream>
2  #include <typeinfo>
3
4  template <typename T>
5  void f(T t){
6   std::cout << "f(" << typeid(T).name() << ")\n";}
7
8  template <typename T>
9  void f(T * t){
10  std::cout << "f(" << typeid(T).name() << "*)\n";
11 }
12
13 int main() { int * p = nullptr;
14     f(0.0);   // odabire f(T), T=double
15     f(0);     // odabire f(T), T=int
16     f(p);     // odabire f(T*), T=int
17     return 0;}
```

Inicijalizacija

```
1  int i;
2  int j = 4;
3  int k(9);
4  int polje[] = {10,15,20,25};
5  double *vec = new double[3]{1.0,2.0,3.0};
6  delete[] vec;
7  int dvodimenzionalno[3][2] = {{1,2},{3,4},{5,6}};
8  int dvodimenzionalno[3][2] = {1,2,3,4,5,6};
9  int dvodimenzionalno[3][4] = {{1},{2},{3}};
10 int dvodimenzionalno[3][4] = { 1, 2, 3 };
```

Uniformna inicijalizacija

```
1 int i{4};
2 int j = {5};
3 //uniformna inicijalizacija
4 //ne dozvoljava konverzije
5 //kod kojih dolazi do
6 //gubitka preciznosti
7 //druge vrste inicijalizacija
8 //dozvoljavaju
```

Uniformna inicijalizacija osigurava korištenje identične sintakse pri inicijalizaciji klasa, STL-a, polja, varijabli itd.

Inicijalizacija listom

U C++11 je dozvoljeno inicijalizirati spremnike listom.

```
1  std::list<double> d{1.1,2.2,3.1,4.2,5.3};
2
3  std::deque<char> e{'a','b'};
4
5  std::set<std::string>skup{"dune","spice","worm"};
6
7  std::map<std::string, std::string> mapa{
8   {"Duke", "Leto"}, {"Lady", "Jessica"},
9   {"Duncan", "Idaho"}};
```

Inicijalizacija listom

Pripaziti na sintaksu:

```
1 std::vector<int> prvi(3); // 3 elementa
2 std::vector<int> drugi(3,1); // 3 elementa
3                              //inicijalizirana s 1
4 std::vector<int> treci{3}; // Inicijalizacija
5                            //jednim elementom
6                            //jednakim 3
7 std::vector<int> cetvrti{3,1}; //2 elementa: 3, 1
```

Inicijalizacijske liste

Koristimo `std::initializer_list<T>` definiranu u zaglavlju `<initializer_list>`. Inicijalizacijska lista predstavlja konstantnu listu argumenata tipa T.

Klasa nudi tri metode:

- `begin()`
- `end()`
- `size()`

```
1  void f(std::initializer_list<int> il) {
2    using Iterator =
3          std::initializer_list<int>::iterator;
4
5   for(Iterator it = il.begin(); it != il.end();
6                                          ++it)
7           std::cout << *it << "␣" ;
8    std::cout << std::endl; }
```

Inicijalizacijske liste

```
1  f({1,2,3,4,5});
2  f({34,56});
3
4  //povratni tip mora dozvoljavati
5  //inicijalizaciju inicijalizacijskom
6  //listom, da bi u return
7  //mogli koristiti inicijalizacijsku
8  //listu
9  std::vector<double> g()
10 {
11   return {0.1,0.2};
12 }
```

Automatska dedukcija tipa: auto

Ključna riječ auto označava da želimo prepustiti prevodiocu da zaključi tip varijable prema **tipu izraza** koji je pridružen toj varijabli.

```
1 int a = 1;
2 double b = 2.2;
3 auto c = a+b; //c je tipa double
```

auto je korisno upotrebljavati s predlošcima jer je tada ponekad teško utvrditi tip rezultata operacija:

```
1 template <typename T1, typename T2>
2 void f(T1 t1, T2 t2){
3  auto t = t1 + t2;
4 }
```

Automatska dedukcija tipa: auto

Postupak dedukcije tipa koristeći auto je ekvivalentan postupku dedukcije tipa pri pozivu generičke funkcije.

```
1 auto x = expr; //1. slucaj
2 const auto x = expr; //1. slucaj
3 auto &x = expr; //2. slucaj
4 auto &&x = expr; //3. slucaj
```

const se ne ignorira kod pokazivača na konstantne objekte:

```
1 int i = 12;
2 const int & k = i;
3
4 auto pk = &k; // const int * pk = &k;
5 auto pi = &i; // int * pi = &i;
```

Automatska dedukcija tipa: auto

Pri korištenju auto, referencu treba eksplicitno navesti (ukoliko želimo da rezultantni tip bude referenca). U tom slučaju se const ne ignorira.

```
1  int i = 12;
2  const int & k = i;
3
4  auto & k2 = k;      // const int & k2 = k;
5  const auto ii = i; // const int ii = i;
6
7  auto v = 0.0, *pv = &v; //mozemo deklarirati
8  //vise auto varijabli u liniji
```

Automatska dedukcija tipa: auto

Ako varijablu inicijaliziramo izrazom u vitičastim zagradama pri korištenju tipa auto, deducirani tip će biti std::initializer_list.

```
1 auto v1{15};   //v1 je tipa int
2 auto v2={15}; //v2 je tipa int
3 //ili std::initializer_list<int>
4 // kod starijih prevodioca
5 auto v3={15,18}; // v3 je tipa
6 //std::initializer_list<int>
```

Automatska dedukcija tipa: `decltype`

`auto` koristimo za dedukciju tipa varijable **samo pri inicijalizaciji varijable**. U svim ostalim slučajevima koristimo `decltype`.

```
1 decltype(f()) sum = 1.0; // sum ima tip koji
2 //vraca f()
```

Ukoliko se koristi `decltype` za dedukciju tipa varijable, tada se **ne ignorira** konstantnost i **ne ignorira** se referenca.

```
1 const double a = 0.0;
2 const double & b = a;
3
4 decltype(a) c = 2.0; // const double c = 2.0;
5 decltype(b) d;      //  const double & d; greska
6                      // neinicijalizirana referenca
```

Automatska dedukcija tipa: `decltype`

Primjenom `decltype` na **izraz** dobivamo lijevu referencu ukoliko izraz ima lijevu vrijednost. Inače dobivamo tip bez reference.

```
1 double a = 1.5, *p = &a, &r = a;
2 decltype(r) b = a; // double & b = a;
3 decltype(*p) c;    // double & c; greska
4                    // neinicijalizirana referenca
5 decltype(*p + 0) d;  // double d;
6 decltype( (a) ) e;     // greska - double & e;
```

- Dereferencirani pokazivač ima lijevu vrijednost stoga `decltype(*p)` vraća referencu.
- Dodavanjem konstante dereferencirnom pokazivaču dobivamo desnu vrijednost, stoga `decltype(*p + 0)` vraća tip.
- Kada varijablu stavimo u oble zagrade dobivamo izraz, stoga `dectype((a))` vraća referencu.

Nova sintaksa funkcije

C++11 dozvoljava sintaksu funkcije u kojoj povratni tip funkcije dolazi nakon funkcijskih argumenata.

```
1 auto f() -> int{
2     int tmp = 3;
3     // ...
4     return tmp; }
```

Navedena sintaksa je korisna kod predložaka funkcije jer se povratni tip može zaključiti koristeći `decltype`.

```
1 template <typename T1, typename T2>
2 auto g(T1 const & t1, T2 const & t2) ->
3                                 decltype(t1*t2){
4         return t1*t2; }
```

Nova sintaksa funkcije

```
1 template <typename C, typename I>
2 auto g(C & spremnik, I indeks) ->
3                         decltype(spremnik[indeks])
4 {
5     return spremnik[indeks];
6 }
```

Deducirani povratni tip će biti `double &`, stoga je sljedeći kod ispravan:

```
1 std::vector<double> vec{1.0,2.0};
2 g(vec,1) = 3.14;
```

Nova sintaksa funkcije

C++14 omogućava sintaksu:

```
1 template <typename C, typename I>
2 decltype(auto) g(C & spremnik, I indeks)
3 {
4     return spremnik[indeks];
5 }
```

range-for petlja

```
for(deklaracija : izraz)
    naredba;
```

`izraz` predstavlja niz elemenata (polje), višedimenzionalno polje, niz u vitičastim zagradama, `string`, `vector` ili bilo koji **STL** spremnik koji ima `begin()` i `end()` metode.

`deklaracija` mora biti takva deklaracija varijable da se element niza može konvertirati u tu varijablu.

```
char niz[] = { 'a','b','c','d','e'};
for(char x : niz) std::cout << x << ",";
```

Ekvivalentno s:

```
for(int i=0; i<5; ++i)  std::cout<<niz[i]<< ",";
```

range-for i auto

U range-for petlji možemo koristiti varijablu tipa auto, što omogućava prevodiocu da automatski odredi potreban tip varijable kojom iteriramo po polju ili spremmniku.

```
1 for(auto x : niz) std::cout << x << ",";
```

Ukoliko želimo mijenjati spremnik, varijablu po kojoj iteriramo treba deklarirati kao **referencu**.

```
1 for(auto& x : niz)  x += 1;
```

```
1 std::vector<double> vec{1.1,2.2,3.3};
2 for(auto x : vec) std::cout << x << ",";
3 for(auto x : {"aaa","bbb","ccc"})
4     std::cout << x << ",";
```

range-for i polja

Kod višedimenzionalnih polja treba definirati redak kao referencu kako bi izbjegli automatsku konverziju niza u pokazivač na prvi element niza:

```
1 int polje[][3] = { {1,3,5}, {2,4,6}, {7,8,9} };
2
3 for(auto& redak : polje){
4   for(auto st : redak) std::cout << st << ",";
5         std::cout << std::endl;
6 }
```

Napomene:

- Dinamički alocirano polje ne može se koristiti u range-for petlji.
- Range-for petlja primijenjena na STL spremnike se ne može koristiti za dodavanje/izbacivanje elemenata u/iz spremnik(a) (invalidiraju se iteratori).

Lambda izrazi

Lambda izrazi su posebni funkcijski objekti (bezimene inline funkcije).

```
[lista varijabli iz okruzenja](lista parametara)
                                -> povratni tip
{ tijelo funkcije }
```

Lambda izraz koji izračunava treću potenciju broja:

```
auto p = [](int i) -> int { return i*i*i; };
int j = p(7);
```

Povratni tip (koji se piše novom sintaksom) **ne moramo specificirati** i tada se **deducira**. C++11 pravila kažu:

- Ako lambda u tijelu ima samo jednu return naredbu, onda se povratni tip deducira iz return izraza.
- Ako je tijelo lambda složenije od jedne return naredbe, onda je povratni tip void.

Lambda izrazi

Od verzije C++14, povratni tip se uvijek deducira `auto` pravilima dedukcije.

Lambda izraz koji izračunava treću potenciju broja:

```
1 auto r = [](double v) {return v*v*v; };
2 // povratni tip se deducira kao double
3 double vr = r(12.0);
```

Lambda izraze najčešće koristimo kao argumente algoritama:

```
1  std::sort(rijeci.begin(), rijeci.end(),
2   [](const std::string & a, const std::string & b)
3                     { return a.size() < b.size(); });
4
5
6  std::for_each(rijeci.begin(), rijeci.end(),
7           [](const std::string & a)
8                   { std::cout << a << std::endl; });
```

Ukoliko lambda izraz ne prima parametre možemo **ispustiti oble zagrade**. Uglate zagrade su **obavezne**.

Hvatanje varijabli iz okruženja (capture)

Lambda izraz može dohvatiti varijable iz svog **lokalnog okruženja** navodeći ih u uglatim zagradama. Varijable se pišu odvojene zarezom. Prijenos varijabli je **po vrijednosti**, odnosno po **referenci** ako se & stavi ispred imena varijable.

```
1 template <typename T>
2 void print(std::ostream & out, std::vector<T>
3                                  const & spremnik){
4  std::for_each(spremnik.begin(), spremnik.end(),
5    [&out](T const & t){ out << t << std::endl; });
6 }
```

Lambda može upotrebljavati varijable iz okruženja mjesta deklaracije (ne poziva). Varijabla koju želimo koristiti mora biti definirana **prije lambde**.

Hvatanje varijabli iz okruženja (capture)

Oblici dohvaćanja varijabli iz okoline definicije izraza:

- [] prazne uglate zagrade znače da lambda ne hvata varijable iz svog okruženja.
- $[x, y, \&z]$ hvata tri varijable, x i y po vrijednosti i z po referenci.
- [&] hvata sve varijable iz svog okruženja po referenci.
- [=] hvata sve varijable iz svog okruženja po vrijednosti.
- $[\&, x, y]$ hvata sve varijable iz svog okruženja po referenci osim x i y koje hvata po vrijednosti.
- $[=, x, y]$ hvata sve varijable iz svog okruženja po vrijednosti osim x i y koje hvata po referenci.

Dohvaćanje varijabli po vrijednosti i referenci

Varijable iz okruženja deklaracije lambda izraza koje su dohvaćene po **vrijednosti** se kopiraju unutar lambda izraza i njene daljnje promjene ne utječu na lambda izraz. Kopija varijable unutar lambda izraza je **konstantna**.

```
1 double pi = 3.14;
2 auto f = [pi]() { pi += 0.001; return pi*pi; };
3 // greska, pi se ne smije mijenjati
```

Ukoliko je kopiju varijable potrebno mijenjati, lambda izraz treba označiti kao `mutable`.

```
1 double pi = 3.14;
2 auto f = [pi]() mutable {
3       pi += 0.001; return pi*pi; };
4 //OK
```

Dohvaćanje varijabli po vrijednosti i referenci

Ako varijablu dohvaćamo preko **reference**, unutar lambda izraza radimo s referencom. Odgovornost je programera da je dohvaćena referenca asocirana s varijablom koja je definirana tijekom rada s lambda izrazom. Varijable dohvaćene preko reference nisu konstantne i njene promjene se vide unutar lambda izraza.

```
1 double pi = 3.14;
2 auto f = [&pi]() { pi += 0.001; return pi*pi; };
3
4 pi = 6;
5 std::cout << f() << std::endl; // ispisuje 36.012
```

init capture

U standardu C++14 hvatanje varijabli iz okruženja je generalizirano na način da se unutar uglatih zagrada može koristiti `var = expr` (ili `&var = expr`).

```
1 auto g = [pi = pi](double x) { return pi*x; };
2 std::cout << g(3.29) << std::endl;
```

Kod `var = expr`, `var` je varijabla unutar dosega lambde, `expr` je izraz unutar okružujućeg dosega. Zato možemo imati isto ime i kod varijable i izraza.

```
1 auto g = [pi = std::move(pi)](double x) {
2 return pi*x; };
3 //premjestamo varijablu pi u lambda
```

Generički lambda izrazi

U dosadašnjim primjerima su argumenti lambda izraza morali biti eksplicitno navedeni. C++14 uvodi mogućnost deklariranja argumenata tipa `auto`. U tom slučaju se vrši standardna `auto` dedukcija parametara.

```
1 std::vector<int> vec{6,7,3,5,2,1,9,0,6};
2 std::sort(vec.begin(), vec.end(),
3                [](auto x, auto y) { return x < y; }
4           );
```