Sveučilište u Zagrebu
PMF – Matematički odsjek

Objektno programiranje (C++)

Predavanja 05 – Generičko programiranje

Vinko Petričević

Predlošci općenito

- C++ dopušta pisanje generičkih funkcija:
- template<class T> void f(T x)
- Za svaki tip za koji pozovemo funkciju f, compiler će *instancirati* funkciju s tim tipom
- Zbog toga i definiciju i implementaciju take funkcije pišemo u .h datoteku
- Ukoliko smo nešto pogriješili u pisanju funkcije, greške najčešće dobijemo tek kada pozovemo funkciju sa konkretnim tipom (općenito se provjere samo semantičke stvari, a tek s pravim tipom zadovoljava li on sve operatore/funkcije)
- Ukoliko želimo da je funkcija inline, inline pišemo iza template
- Ukoliko imamo više parametara istog tipa, npr.
  template<class T> T min(T a, T b)
  i pozovemo funkciju s istim parametrima sve će biti OK. Međutim pozovemo li funkciju npr. min(1,1.5) compiler neće znati što je T, pa onda trebamo naglasiti npr. min<double>(1, 1.5)
- Ukoliko za neki tip imamo posebnu implementaciju, možemo pisati specijalizaciju, npr.
  template<> const char* min(const char* a, const char* b)

- Primjer – sort
- Možemo ubaciti i ispis

Netipski parametri

- Možemo napraviti operator<< ispisa na stream
- Tip T ne mora uvijek biti klasa, može biti i broj, npr.
  Za int pi[7], funkcija koja bi primila pi, trebala bi imati izgledati:
  void ispis(int(&p)[7])

- odnosno, za ispis niza proizvoljne duljine:
  template<long N> void ispis(int(&p)[N]) …

- odnosno proizvoljnog polja:
  template<typename T, long N> void ispis(T(&p)[N]) …

- Ukoliko bi napravili operator<< za proizvoljno polje, to bi nam vjerojatno izazvalo više grešaka zbog ispisa niza znakova

- Možemo napraviti npr. funkciju koja računa faktorijel za vrijeme compiliranja programa:
  template<size_t N> long long factorial() {return N * factorial<N - 1>();}
  template<> long long factorial<1>() {return 1;}
  …
  std::cout << factorial<7>() << std::endl;

Više parametara

- Svaki predložak može primati proizvoljan broj parametara (unaprijed zadan), npr.
  template<class S, class T, class U> void f(S x, S y, T z, T u, U v)...

- Ukoliko želimo napraviti specijalizaciju, možemo za jedan ili više tipova – primjer specijalizacija.cpp

- Od standarda 11 dopušteno je pisati i
  template <typename... Args>
  pogledati primjer rekIspis.cpp

- Moguće je napraviti predložak kojem će nekoliko posljednjih parametara imati defaultne vrijednosti ukoliko su ispušteni – primjer default.cpp

Predlošci klasa

- C++ nam dopušta i da cijele strukture parametriziramo.

```
template <class T> struct stog {
    T podaci[100];
    int vrh;
    stog() { vrh = 100; }
    T top() { ... return podaci[vrh]; }
    void push(T data);
};
template<class T>
    void stog<T>::push(T data) {
            ...
      }
```

- Za razliku od funkcijskih predložaka, argumenti predloška strukture ne mogu biti deducirani iz konteksta, pa ih treba navesti

```
stog<int> stogIntova;
stog<string> stogStringova;
```

- I dalje te strukture koristimo kao i svaku drugu:

```
stogIntova.push(10);
stogStringova.push("op");
```

- Na isti način smo koristili i STL-containere:

Predlošci klasa - friend

- Pogledajmo primjer queue1.cpp
- U predlošku Queue pristupamo privatnom dijelu klasu QueueItem, pa u toj klasi trebamo navesti da joj je klasa Queue prijateljska

- Ukoliko bismo željeli napraviti operator<< za Queue, trebamo navesti da je i taj operator prijateljska funkcija – pogledajmo primjer queue2.cpp

- Elegantnije bi bilo napraviti QueueItem kao podklasu klase Queue, onda ne moramo pisati QueueItem<T> - pogledajmo queue3.cpp

- Pogledati primjer parcijalne specijalizacije za pointere – isPointer.cpp

Primjer – ispis spremnika

- Promotrimo sljedeći operator za ispis proizvoljnog vectora:

```
template<class T>
std::ostream& operator<<(std::ostream& o, const std::vector<T>& v) {
    o << "{";
    if (!v.empty()) {
        typename std::vector<T>::const_iterator i = v.begin();
        o << *i;
        while (++i != v.end())
            o << ',' << *i;
    }
    o << "}";
    return o;
}
```

- Kako bismo ga mogli preraditi da ispisuje proizvoljni spremnik, kao npr. sljedeća (C++11) funkcija:

```
template<class T> void ispisi(const T& v) {
    std::cout << "{";      bool first = true;
    for (auto i : v) {
        if (first) first = false; else std::cout << ",";    std::cout << i;
    }   std::cout << "}"; }
```

Template-template

- vector<T> je ustvari tipa vector<T, std::allocator<T>>, pa možemo koristiti:

```
template<template<class, class> class Cont, class T, class A>
void ispisi(const Cont<T, A> & k){
    typename Cont<K, A>::const_iterator i = k.begin();
    ...
}
```

- Sada možemo pisati:

```
template< template<class, class> class Cont, class T, class A>
std::ostream& operator<<(std::ostream& o, const Cont<T, A>& v) {
 ...
}
```

- Ukoliko želimo ispisati proizvoljni set (on ima 3 parametra, tip, predikat i alokator)

```
template<template<class, class, class> class Cont, class K, class P, class A>
std::ostream & operator<<(std::ostream & o, const Cont<K, P, A>& v) {
 ...
        typename Cont<K, P, A>::const_iterator i = v.begin();
 ...
}
```

Template-template

- Ukoliko bi željeli ispisivati i mape, mogli bi pokušati (mapa ima 4 parametra)

```
template<class S, class T>
std::ostream& operator<<(std::ostream& o, const std::pair<S, T>& p) {
    o << "(" << p.first << "," << p.second << ")";
    return o;
}

template< template<class, class, class, class> class Cont, class K, class V,
 class P, class A>
std::ostream& operator<<(std::ostream& o, const Cont<K, V, P, A>& v) {
 ...
}
```

međutim, dobili bismo greške sa dvosmislenošću (za razliku od funkcije ispisi sa prethodnih slideova, koja bi radila dobro)

- Ako bismo izbacili operator<< za set, sve bi radilo dobro
- Problem je u tome što je i ostream klasa koja je parametrizirana sa 2 parametra

Template-template

- Isto kao što možemo napraviti funkciju koja prima dva prizvoljna spremnika (s dva parametra):

```
template< template<class, class> class Cont, class T, class A, template<clas
s, class> class Cont1, class T1, class A1>
void f(const Cont<T, A>& v, const Cont1<T1, A1>& v1) {
    std::cout << v;    std::cout << v1;
}
```

ili funkciju g kojoj bi mogli poslati opći ostream (koja ispiše 1 ako pozovemo g(std::cout)):

```
template< template<class, class> class Cont, class T, class A> void g(Cont<T
, A>& s) {
    s << "1\n";
}
```

mogli bismo pisati:

```
template<template<class, class> class ostr, class C, class CT, template<clas
s, class, class> class Cont, class K, class P, class A>
ostr<C, CT>& operator<<(ostr<C,CT>& o, const Cont<K, P, A>& v) {
 ...
}
```

i nakon ovoga bi std::cout << s radilo za proizvoljni set s, a i ispis mape bi bio dobar

Slični templateovi

- Slično kao što bismo mogli napraviti
  tempalate<class S, class T> S operator+(const S&, const T &) koji će zbrojiti dva tipa S i T na kojima postoji operator+, mogli bismo slične stvari raditi i sa klasama koje su napravljene na tipovima koji imaju srodne operacije
- Promotrimo klasu X iz datoteke slicni0.cpp
- Želimo npr. omogućiti operacije na *sličnim* tipovima (npr. zbrajanje intova i doubleova)
- Da je varijabla i bila javna, radilo bi npr (bilo koje od sljedećeg):
  template<class S> X operator+(const X<S>& d) const { return X(i+d.i);}
  template<class S> friend X operator+(const X&l, const X<S>& d) { return X(l.i+d.i);}

- Ako nije javna, u prvom slučaju bi trebali dodati da su npr. sve klase X prijateljske (slicni1.cpp), a u drugom npr. razdvojiti implementaciju od definicije funkcije (slicni2.cpp)
- Općenito razdvajanje definicije od implementacije funkcije s jednim tipom parametara bismo mogli napraviti i kao u slicni3.cpp

- Ukoliko samo neku funukciju želimo napraviti prijateljskom, možemo kao u templ-fr0.cpp. U tom slučaju su nam sve funkcije ispisi prijateljske, tj. možemo i iz jedne pristupati privatnim članovima drugog templatea, kao što se vidi u templ-fr1.cpp
- Ukoliko to baš ne želimo dopustiti, možemo napraviti da je templateu sa određenim tipom samo *njena* funkcija ispiši (sa tim tipom) prijateljska, kao što je napravljeno u templ-fr2.cpp