Sablonok (template-ek) C++-ban

Pataki Norbert

Programozási Nyelvek és
Fordítóprogramok Tanszék

Programozási Nyelvek (C++)

Témák

Megkerülések

- `int printf( const char* format, ... );`
- `void free( void* ptr );`
- `void* malloc( int bytes );`
- `#define MAX(a,b) ((a)<(b)?(b):(a))`
- Típusinformációk?
- Problémák

Sablonok

- Kódrészlet, típusparaméterekkel ellátva
- Különböző típusokkal használható
- Akár olyan típussal is, ami a sablon megírásakor nem létezik
- Függvény sablonok (template)
- Osztály sablonok (template)
- Típusbiztonság

Példa – függvény sablon

```
template <class T>
T max( T a, T b )
{
  return a < b ? b : a;
}
```

Példa – függvény sablon (jobb megoldás)

```
template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}
```

- Több típussal használható
- Elkerüljük felesleges másolásokat, (pl. `std::vector`-ral használva)

C++ template-ek működése

tmpmax.cpp:

```
template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}

int main()
{
}
$ g++ -c tmpmax.cpp
$ nm tmpmax.o
0000000000000000 T main
```

C++ template-ek működése

```
template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}
int main()
{
  int x = max( 4, 1 );
  int y = max( x, 7 );
  double d = max( 3.2, 7.4 );
}
$ g++ -c tmpmax.cpp
$ nm tmpmax.o
0000000000000000 T main
0000000000000000 W _Z3maxIdERKT_S2_S2_
0000000000000000 W _Z3maxIiERKT_S2_S2_
```

Példányosítás, paraméter dedukció

- A template-eket példányosítani kell. Nem lehet azelőtt lefordítani.
- Példányosítás nélkül: nincs alacsony-szintű megfelelője.
- Példányosítás nélkül: általában nem derülnek ki a fordítási hibák sem.
- Template-ek: általában header file-ban írjuk meg (az osztály sablonokat is).
- Példányosítás: template paraméter helyére a konkrét típusparaméter kerül.
- Példányosítás: minden különböző típussal, amivel használjuk
- Paraméter dedukció: függvény sablonok esetén a fordító általában le tudja vezetni a hívás paramétereiből a sablon paramétereket.

Problémák

```
#include <iostream>

template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}
int main()
{
  std::cout << max( "alma", "citrom" );
}
```

Fordítási hiba

```
tmpmax.cpp: In function 'int main()':
tmpmax.cpp:11:38: error: no matching function for call to 'max(const char [5], const char [7])'
   std::cout << max( "alma", "citrom" );
                                      ^
tmpmax.cpp:4:10: note: candidate: template<class T> const T& max(const T&, const T&)
 const T& max( const T& a, const T& b )
          ^
tmpmax.cpp:4:10: note:   template argument deduction/substitution failed:
tmpmax.cpp:11:38: note:   deduced conflicting types for parameter 'const T' ('char [5]' and 'char [7]')
   std::cout << max( "alma", "citrom" );
                                      ^
```

Problémák

```
#include <iostream>

template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}
int main()
{
  std::cout << max( 7.8, 12 );
}
```

Ez is fordítási hibát okoz.

Problémák

```
//...

int main()
{
//  std::cout << max( "hagyma", "citrom" )
//            << std::endl;
  std::cout << max( "citrom", "hagyma" )
            << std::endl;
}

$ g++ tmpmax.cpp
$ ./a.out
citrom
```

Problémák

```
//...

int main()
{
  std::cout << max( "hagyma", "citrom" )
            << std::endl;
  std::cout << max( "citrom", "hagyma" )
            << std::endl;
}

$ g++ tmpmax.cpp
$ ./a.out
hagyma
hagyma
```

Explicit specializáció

```
std::cout << max<double>( 1.7, 8 );
std::cout <<
  max<std::string>( "hagyma", "citrom" ); // hagyma
```

Nincs paraméter dedukció

Elvárások megsértése

```
// ...

class complex
{
  // ....
};

void f()
{
  complex a( 3.2, 4.3 );
  complex b( 1.7, 1.2 );
  complex c = max( a, b );
}
```

Fordítási hiba

```
$ g++ -c tmpmax.cpp
tmpmax.cpp: In instantiation of ‘const T& max(const T&, const T&) [with T = complex]’:
tmpmax.cpp:20:25:   required from here
tmpmax.cpp:6:12: error: no match for ‘operator<’ (operand types are ‘const complex’
                                                  and ‘const complex’)
   return a < b ? b : a;
          ^
```

Megszorítások

- Az előző példában nem jeleztük `T`-ről, hogy rendezhetőnek kell lennie.
- Csak a sablon törzsében jeleztük.
- C++ sablonok: megszorítás nélküli paraméterek
- Fordítási hiba, ha megsértjük.
- Nehezebben érthető példa:

```
#include <list>
#include <algorithm>
//...
std::list<int> c;
std::sort( c.begin(), c.end() );
// c.sort();
```

Fordítási hiba

```
In file included from /usr/include/c++/5/algorithm:62:0,
                 from listsort.cpp:2:
/usr/include/c++/5/bits/stl_algo.h: In instantiation of 'void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = std::_List_iterator<int>; _Compare = __gnu_cxx::__ops::_Iter_less_iter]':
/usr/include/c++/5/bits/stl_algo.h:4698:18:   required from 'void std::sort(_RAIter, _RAIter) [with _RAIter = std::_List_iterator<int>]'
listsort.cpp:7:33:   required from here
/usr/include/c++/5/bits/stl_algo.h:1964:22: error: no match for 'operator-' (operand types are 'std::_List_iterator<int>' and 'std::_List_iterator<int>')
     std::__lg(__last - __first) * 2,
                      ^
In file included from /usr/include/c++/5/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/5/list:60,
                 from listsort.cpp:1:
/usr/include/c++/5/bits/stl_iterator.h:328:5: note: candidate: template<class _Iterator> typename std::reverse_iterator<_Iterator>::difference_type std::operator-(const std::reverse_iterator<_Iterator>&, const std::reverse_iterator<_Iterator>&)
     operator-(const reverse_iterator<_Iterator>& __x,
     ^
/usr/include/c++/5/bits/stl_iterator.h:328:5: note:   template argument deduction/substitution failed:
In file included from /usr/include/c++/5/algorithm:62:0,
                 from listsort.cpp:2:
/usr/include/c++/5/bits/stl_algo.h:1964:22: note:   'std::_List_iterator<int>' is not derived from 'const std::reverse_iterator<_Iterator>'
     std::__lg(__last - __first) * 2,
                      ^
In file included from /usr/include/c++/5/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/5/list:60,
                 from listsort.cpp:1:
/usr/include/c++/5/bits/stl_iterator.h:385:5: note: candidate: template<class _IteratorL, class _IteratorR> typename std::reverse_iterator<_Iterator>::difference_type std::operator-(const std::reverse_iterator<_Iterator>&, const std::reverse_iterator<_IteratorR>&)
     operator-(const reverse_iterator<_IteratorL>& __x,
     ^
/usr/include/c++/5/bits/stl_iterator.h:385:5: note:   template argument deduction/substitution failed:
In file included from /usr/include/c++/5/algorithm:62:0,
                 from listsort.cpp:2:
/usr/include/c++/5/bits/stl_algo.h:1964:22: note:   'std::_List_iterator<int>' is not derived from 'const std::reverse_iterator<_Iterator>'
     std::__lg(__last - __first) * 2,
                      ^
```

Példa: STL algoritmus

```
template <class InIt, class Fun>
Fun for_each( InIt first, InIt last, Fun f )
{
  while ( first != last )
  {
    f( *first++ );
  }
  return f;
}
```

A template paraméterek csak a nevükben térnek el!

Akkor is, ha teljesen különböző funkciókat látnak el.

Elvárások

- `InIt`:
  - `operator*`
  - `operator++(int)`
  - `operator!=`
  - Copy konstruktor
- `Fun`:
  - Copy konstruktor
  - Függvényhívás operátor

Milyen típusokkal használható?

- `InIt`:
  - `int*` – az STL algoritmusok használhatóak tömbökkel
  - `std::list<std::string>::iterator`
  - STL-es konténerek iterátora
  - Még nem létező adatszerkezetek iterátora
- `Fun`:
  - Globális (megfelelő típusú) függvényre mutató pointerek
  - Felhasználói típusok, (megfelelő típusú) `operator()`-ával.

A template paraméter lehet

- Típus
- Integrális konstans (pl. `int`, `bool`)
- Külső szerkesztésű objektum vagy függvény címe
- (Nem túlterhelt) tagra mutató pointer

Tömbök átvétele paraméter dedukcióval

```
template <class T, int N>
void print_size( T ( &a )[ N ] )
{
  std::cout << N << std::endl;
}

// ...

int v[] = { 8, 1, 6, 2, 3 };
print_size( v ); // 5
```

Osztály template példák

- `std::vector<double>`
- `std::list<complex>`
- `std::set<int>`
- stb.
- Explicit specializáció
- Nincs paraméterdedukció
- `make_...` utility-k

Funktor példa

```
#include <iostream>

template <class T>
class TPrint
{
  std::ostream& os;
public:
  TPrint( std::ostream& o ): os( o ) { }

  void operator()( const T& t )
  {
    os << t << ' ';
  }
};
```

Funktor példa

```
std::set<std::string> v;
std::deque<double> d;

// ...

std::for_each( v.begin(),
               v.end(),
               TPrint<std::string>( std::cout ) );

std::for_each( d.begin(),
               d.end(),
               TPrint<double>( std::cerr ) );
```

Típusbiztonság

Funktor példa

```
#include <iostream>

class Print
{
  std::ostream& os;
public:
  Print( std::ostream& o ): os( o ) { }

  template <class T>
  void operator()( const T& t )
  {
    os << t << ' ';
  }
};
```

Funktor példa

```
std::set<std::string> v;
std::deque<double> d;

// ...

std::for_each( v.begin(),
               v.end(),
               Print( std::cout ) );

std::for_each( d.begin(),
               d.end(),
               Print( std::cerr ) );
```

Típusbiztonság

Lusta példányosítás

```
#include <set>
#include <list>

class complex
{
  // ...
};

int main()
{
  std::list<complex> a; // OK
  a.push_back( complex( 1.2, 3.4 ) ); // OK
  a.sort(); // Ford. hiba: nincs operator<

  std::set<complex> b; // OK
  b.insert( complex( 1.2, 3.4 ) ); // Ford. hiba:
                                   // nincs operator<
}
```

Típusok ekvivalenciája

```
template <class T, int N>
class Array
{
  T v[ N ];
};

// ...

Array<int, 10> s;
Array<int, 5 * 2> t = s; // OK
Array<int, 12> a = s; // Ford. hiba
```

Példa

```
struct Base
{
  virtual ~Base() { }
  virtual void f() const = 0;
};
void g( const Base* p )
{
  p->f();
}
// ----
template <class T>
void tf( const T& t )
{
  t.f();
}
```

Öröklődés vagy template

- Hasonlóság: olyan előre megírt kódrészlet, ami alkalmazkodni tud olyan kódokhoz, ami még nem ismert/megírt.
- Öröklődés:
  - Típusbiztonság
  - Rugalmasság: elég futási időben tudni, hogy mi hívódik meg
  - Lefordítható előre, de nagyobb futási idő
- Template:
  - Típusbiztonság
  - Hatékonyság: fordításkor tudja a fordító, hogy mi hívódik meg
  - Rugalmatlanság: a példányosítást a fordító végzi el, nem függhet futási idejű adatoktól
  - Egymástól független típusok