Concept C++

Template-k a jelenlegi szabványban

- A C++ template rendszere típus nélküli
- Nincs nyelvi támogatás a sablon-szerződés modell alkalmazására
- Az esetleges hibák a példányosítási folyamatban túl későn derülnek ki =>

  Hosszú, bonyolult, nehezen értelmezhető hibaüzenetek

Példa

```
...
std::list<int> intList;
intList.push_back(42); ... intList.push_back(5);
...
std::sort(intList.begin(), intList.end());
```

Ezt gcc-vel fordítva:

Hibaüzenet

```
/usr/include/c++/4.1.3/bits/stl_algo.h: In function ‘void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<int>]’:
templ.cpp:13:   instantiated from here
/usr/include/c++/4.1.3/bits/stl_algo.h:2713: error: no match for ‘operator-’ in ‘__last
    - __first’
/usr/include/c++/4.1.3/bits/stl_algo.h: In function ‘void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator)
    [with _RandomAccessIterator = std::_List_iterator<int>]’:
/usr/include/c++/4.1.3/bits/stl_algo.h:2714:   instantiated from ‘void

    std::sort(_RandomAccessIterator,  _RandomAccessIterator) [with

    _RandomAccessIterator = std::_List_iterator<int>]’
templ.cpp:13:   instantiated from here
```

Hibaüzenet folyt.

```
/usr/include/c++/4.1.3/bits/stl_algo.h:2360: error: no match for ‘operator+’ in ‘__first
    + 16’
/usr/include/c++/4.1.3/bits/stl_algo.h: In function ‘void
    std::__insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<int>]’:
/usr/include/c++/4.1.3/bits/stl_algo.h:2363:   instantiated from ‘void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator)
    [with _RandomAccessIterator = std::_List_iterator<int>]’
/usr/include/c++/4.1.3/bits/stl_algo.h:2714:   instantiated from ‘void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<int>]’
templ.cpp:13:   instantiated from here
/usr/include/c++/4.1.3/bits/stl_algo.h:2273: error: no match for ‘operator+’ in ‘__first
    + 1’
```

Hibaüzenet vége

```
/usr/include/c++/4.1.3/bits/stl_algo.h:2363:   instantiated from ‘void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator)
    [with _RandomAccessIterator = std::_List_iterator<int>]’
/usr/include/c++/4.1.3/bits/stl_algo.h:2714:   instantiated from ‘void

    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with

    _RandomAccessIterator = std::_List_iterator<int>]’
templ.cpp:13:   instantiated from here
/usr/include/c++/4.1.3/bits/stl_algo.h:2279: error: no match for ‘operator+’ in ‘__i +
    1’
```

A hiba oka

- Ezt látva joggal gondolhatunk arra, hogy az stl implementációban van a hiba
- **Valójában: a sort random access iterator-t vár, de a list csak forward iteratort szolgáltat**
- Erre azonban csak ez a sor utal:

  templ.cpp:13: instantiated from here
  /usr/include/c++/4.1.3/bits/stl_algo.h:2273: error: no match for ‘operator+’ in ‘__first + 1’

Az új szabány

- A problémát a C++ szabványbizottság is látja
- A következő szabványban bevezetik a **concept checking** fogalmát
- Lehetővé válik a sablon-szerződés modell nyelvi szinten való alkalmazása
- Cél: rövid, érthető hibaüzenetek

Az új nyelvi elemek

- Három új nyelvi konstrukcióval bővül a C++
- Concept:
  - A követelmények leírására szolgál
- Concept_map:
  - Szemantikus kapcsolat egy konkrét típus és egy konkrét Concept között
- Requires
  - Megadja, hogy egy adott template függvény vagy osztály esetében a template paraméter milyen Concept-eknek kell, hogy megfeleljen

Concept

- Olyan típusok és függvényszignatúrák halmaza, amelyeket a template függvényünkben vagy osztályunkban használni szeretnénk.
- pl. LessThanComparable concept:

```
concept LessThanComparable<typename T>
{
    bool operator<(T a, T b);
};
```

Requires

- A LessThanComparable concept alkalmazása egy template algoritmuson:

```
template<typename T>
requires LessThanComparable<T>
T min(const T& a, const T& b)
{
  return a < b ? a : b;
};
```

Requires egyszerűbben

- Ha egy concept csak egy típusra definiál megszorításokat (mint ahogy az előbbi példában), akkor egyszerűbben is definiálhatjuk a template függvényünket, osztályunkat:

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

- Azaz a typename kulcsszó helyére közvetlenül beírhatjuk a concept nevét.

Összetettebb példa a Concept-re

```
concept Regular<typename T> {
  // default constructor
  T:: T();
  // copy constructor
  T:: T(const T&);
  // destructor
  T::~T();
  // copy assignment
  T& operator=(T&, const T&);
  // equality comparison
  bool operator==(const T&, const T&);
  // inequality comparison
  bool operator!=(const T&, const T&);
  // swap
  void swap(T&, T&);
};
```

Multi-Type Concept

- Egy concept több típuson is definiálhat követelményeket:

```
concept Convertible<typename T, typename U>
{
   operator U(const T&);
};
```

- Illetve lehet default értéke a típusváltozóknak:

```
concept EqualtyComparable<typename T, typename U = T>
{
   bool operator==(T, U)
};
```

Összetett példa Multi-Type Concept-re

```
concept InputIterator <typename Iter, typename Value> {
  Iter :: Iter ();                                    // default constructor
  Iter :: Iter (const Iter &);                        // copy constructor
  Iter ::~ Iter ();                                   // destructor
  Iter & operator=(Iter&, const Iter&);               // copy assignment
  Value operator*(const Iter&);                       // dereference
  Iter & operator++(Iter&);                           // pre- increment
  Iter operator++(Iter&, int);                        // post- increment
  bool operator==(const Iter&, const Iter&);          // equality comparison
  bool operator!=(const Iter&, const Iter &);         // inequality comparison
  void swap(Iter &, Iter &);                          // swap
};
```

Concept-ek kompozíciója

- Lehetőségünk van a korábban definiált concept-eket felhasználni újabbak definiálásához
- Ezáltal az összetett concept-ek definíciója egyszerűsödik
- Ennek két módja van:
  - beágyazás
  - finomítás

Példa a beágyazásra

- A Regular concept-et felhasználjuk az InputIterator concept definiálásakor:

```
concept InputIterator<typename Iter, typename Value>
{
  requires Regular<Iter>;

  Value operator*(const Iter&); // dereference
  Iter& operator++(Iter&);      // pre-increment
  Iter operator++(Iter&, int);  // post-increment
};
```

Példa a finomításra

- Szintén a Regular concept-et használjuk fel:

```
concept InputIterator<typename Iter,typename Value> :
    Regular<Value>
{
  Value operator*(const Iter&); // dereference
  Iter& operator++(Iter&);      // pre-increment
  Iter operator++(Iter&, int);  // post-increment
};
```

- Elv: Ha van hierarchikus kapcsolat akkor finomítás, egyébként beágyazás
- Különbség: concept_map-ek definiálása (később)

Egy InputIterator-t használó template fv.

```
template<typename Iter, Regular V>
requires InputIterator<Iter, V>

void print(Iter first, Iter last)
{
    while (first != last)
    {
        std::cout << *first++;
    }
    std::cout << std::endl;
}
```

Az igazi InputIterator concept

- Valójában az InputIterator a Value típuson kívül még definiál egy Reference, egy Pointer és egy Difference típust is:

```
concept InputIterator
<typename Iter, typename Value, typename Reference
 typename Pointer, typename Difference>
{
  requires Regular<Iter>;
  requires Convertible<Reference, Value>;

  Reference operator*(const Iter&); // dereference
  Iter& operator++(Iter&);          // pre-increment
  Iter operator++(Iter&, int);      // post-increment
  // ...
}
```

Egy InputIterator-t használó template fv.

- Ezeket a template paramétereket mindig fel kell sorolni

```
template<typename Iter, Regular V, typename R,
          typename P, typename D>
requires InputIterator<Iter, V, R, P, D>

void print(Iter first, Iter last)
{
   while (first != last)
   {
      std::cout << *first++;
   }
   std::cout << std::endl;
}
```

Associated Types

- Egy kényelmesebb megoldás az ún. Associated típusok használata:

```
concept InputIterator<typename Iter> {
  typename value type;
  typename reference;
  typename pointer;
  typename difference type;
  requires Regular<Iter>;
  requires Convertible<reference type, value type>;
  reference operator*(const Iter&);
  Iter& operator++(Iter&);
  Iter operator++(Iter&, int);
  // ...
};
```

Egy InputIterator-t használó template fv.

- Az előbbi megoldással elértük, hogy csak egy argumentuma legyen az InputIterator concept-nek, így a megfelelő template fv.-eket is egyszerűbben definiálhatjuk:

```
template<InputIterator Iter>
requires Regular<Iter::value type>
void print(Iter first, Iter last){
    while (first != last) {
        std::cout << *first++;
      }
    std::cout << std::endl;
}
```

Concept_map

- Szemantikus kapcsolatot biztosít egy konkrét concept és egy konkrét típus között:
- pl. keressük a legrosszabb átlagú diákot

```
struct Diak
{
  std:string name;
  double atlag;
};
...
std::list<Diak> l;
...
std::cout << (*std::min_element(l.begin(), l.end())).name << std::endl;
```

Concept_map 2

- Ez a program nem fordul le, ugyanis a Diak típuson nincs definiálva a < művelet.
- Ezt a concept_map segítségével oly módon meg tudjuk oldani, hogy ne kelljen hozzányúlni a Diak típushoz:

```
concept_map LessThanComparable<Diak>{
  bool operator<(Diak a, Diak b){
    return a.atlag < b.atlag;
  }
};
```

Concept_map 3

- Ha pl. egy X típuson definiált a < operátor, és nem szeretnénk felüldefiniálni, akkor elég az alábbi concept_map-et írni:

```
concept_map LessThanComparable<X>{}
```

- Abban az esetben, ha a concept definíciója elé odaírjuk az auto kulcsszót, akkor a fenti esetben nem kell semmilyen concept_map-et definiálni.

További példák Concept_map-re

- Tfh. az előzőekben az InputIterator conceptet autonak definiáltuk.
- Ekkor lehetséges az alábbi részleges concept_map definíció:

```
concept map InputIterator<char*> {
  typedef char            value type ;
  typedef char&            reference ;
  typedef char*            pointer ;
  typedef std:: ptrdiff t  difference type ;
};
```

- A fenmaradó követelményeket (konstruktor, destruktor stb. a fordító implicit definiálja

További példák Concept_map-re

- Abban az esetben ha az implicit definíció nem helyes definiálhatunk más működést is:
- pl. int-eket szeretnénk input iterátorként használni:

```
concept map InputIterator<int> {
   typedef int value type;
   typedef int reference;
   typedef int* pointer;
   typedef int difference type;
   int operator*(int x) { return x; }
};
```

- Ezzel pl. inicializálhatunk konténereket:

```
std::copy(1, 101, v.begin());
```

Concept_map template

- A concept és a concept_map maguk is lehetnek templatek
- Ugyanis minden pointer (nemcsak a char*) egyben input iterátor is. Ezt egyszerűen le tudjuk írni:

```
template<typename T>
concept_map InputIterator<T*> {
  typedef T  value type ;
  typedef T& reference ;
  typedef T∗ pointer ;
  typedef std:: ptrdiff t difference type ;
};
```

Példa a container adaptorra

- A concept, concept_map konstrukciókkal wrapper osztályok nélkül meg tudunk valósítani container adaptorokat

```
concept Stack<typename X> {
   typename value type;
   void push(X&, const value type&);
   void pop(X&);
   value type top(const X&);
   bool empty(const X&);
};
```

És a hozzá tartozó concept_map

```
template<typename T>
concept_map Stack<std::vector<T> >
{
   typedef T value type;
   void push(std::vector<T>& v, const T& x){
      v.push_back(x);
   }
   void pop(std:: vector <T>& v){
      v.pop_back();
   }
   T top(const std::vector <T>& v){
      return v.back();
   }
   bool empty(const std::vector <T>& v){
      return v.empty();
   }
};
```

Concept-based Overloading

- Mindig az a template fog példányosulni, ahol az aktuális template paraméterek a leginkább illeszkednek a concept-hez:
- Ezáltal az stl sok függvényét, mint pl. az advance sokkal elegánsabban és egyszerűbben definiálhatjuk.

Az advance fv. definíciói

```
template<InputIterator Iter>
void advance(Iter& x, Iter::difference_type n) {
  while (n > 0) { ++x; --n; }
}

template<BidirectionalIterator Iter>
void advance(Iter& x, Iter::difference_type n) {
  if (n > 0) while (n > 0) { ++x; --n; }
  else while (n < 0) { --x; ++n; }
}

template<RandomAccessIterator Iter>
void advance(Iter& x, Iter::difference_type n) {
  x += n;
}
```

Negative constraint

```
template<EqualityComparable T>
class dictionary {
   // slow, linked-list implementation
};

template<LessThanComparable T>
requires !Hashable<T>
class dictionary<T> {
   // balanced binary tree implementation
};

template<Hashable T>
class dictionary<T> {
   // hash table implementation
};
```

AOP szimulálása concept-ekkel

- Tfh. van egy *My* osztályunk, aminek van egy *f* és egy *g* függvénye
- Szeretnénk loggolni valamilyen információt minden f és g függvényhívás előtt és után
- Az AOP paradigmát szimulálhatjuk conceptek és concept_map-ek segítségével

Log concept és concept_map

```
concept Log<typename T>{
   void f();
   void g();
}

concept_map Log<My>{
   void f(My& m){
      //log
      m.f();
      //log
   }
   void g(My& m){
      //log
      m.g();
      //log
   }
}
```

A hibaüzenet

- Az első példaprogramunkra:

```
...
std::list<int> intList;
intList.push_back(42); ... intList.push_back(5);
...
std::sort(intList.begin(), intList.end());
```

Az alábbi hibaüzenetet kapjuk:

```
listconc.cpp: In function ‘int main()’:
listconc.cpp:9: error: no matching function for call to ‘sort(std::_List_iterator<int>,
    std::_List_iterator<int>)’
/home/lupin/local/conceptgcc/bin/../lib/gcc/i686-pc-linux-
    gnu/4.3.0/../../../../include/c++/4.3.0/bits/stl_algo.h:2874: note: candidates are: void
    std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>] <where clause>
listconc.cpp:9: note:   no concept map for requirement
    ‘std::MutableRandomAccessIterator<std::_List_iterator<int> >’
```