Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

GTL
Graphical Template Library

Vatai Emil
V. éves Programtervező Matematikus hallgató
Témavezető:

Dr. **Porkoláb Zoltán** egyetemi docens
Eötvös Loránd Tudományegyetem - Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Budapest, 2007. június 6.

Kivonat

A Graphical Template Library (GTL) [2], egy olyan kísérleti generikus C++ sablon-könyvtár, ami a C++ Standard Template Library (STL) alapötletét próbálja átvinni a grafikus feladatok körébe. A GTL-t 1999-ben készítette el Kisteleki Róbert (ELTE programtervező matematikus hallgató) és Dr. Porkoláb Zoltán.

Eredetileg, a feladat, egy rajzoló program megvalósítása lett volna, amely a GTL-re támaszkodik és a GTL által nyújtott alakzatokat tudja kirajzolni és ezeken a GTL algoritmusait, műveleteit tudja végrehajtani. A program fejlesztése során, kizárólag a generikus programozás eszközeire kellet használni, az öröklődést és a virtuális függvények használatát teljesen mellőzve. A program elkészítése folyamán, komoly problémákba ütköztünk, melyeket a GTL, vagyis általánosan, a (szigorúan) generikus programozás korlátai okoztak. Így a feladat bővült, ezen problémák és korlátok feltárásával és annak a bizonyításával, hogy ezek a korlátok, az elvárt megszorítások (dinamikus polimorfizmus mellőzése) mellet nem kerülhetőek meg.

Végül adunk egy becslést arra, hogy mi lenne az kompromisszum az objektum orientált és a generikus programozás között, amely él mindkét paradigma eszközeivel és a legjobb eredményt nyújtva, azaz hogyan kellene enyhíteni a feltételeket, hogy egy kellően hatékony és elegáns megoldást kapjunk.

Tartalomjegyzék

1. A Generikus programozás kialakulása

A C++ első kereskedelmi forgalomba hozatalakor, 1985-ban még nem támogatta a a generikus programozást, mivel nem volt benne sablon mechanizmus. A sablonokat csak később építették be a nyelvbe, ezzel lehetővé téve a generikus programozás kialakulását.

1.1. Egy kézenfekvő megoldás – öröklődés

Ha az ember geometriai alakzatokat rajzoló és manipuláló programot akar írni, akkor azt objektum orientált megközelítés, vagyis az öröklődés jó választásnak tűnik. A tipikus példa, az öröklődés és az objektum orientált programozás szemléltetésére pontosan az ilyen rajzoló programok implementációja. Egy `Shape` bázis-osztályból indulunk ki és ebből származtatjuk a `Polygon`, `Circle` és hasonló osztályokat. A `Shape` osztályban deklaráljuk a virtuális `draw`, `move`, `rotate`, stb. metódusokat, majd a származtatott osztályokban definiáljuk őket. Egy `Shape*` típusú, azaz alakzatokra mutató mutatókat tároló tömbbe pakoljuk a rajzlap tartalmát, amit egy `DrawAll` függvénnyel rajzolnák ki, amely a tömb minden elemére meghívná a megfelelő alagzat virtuális `draw` metódusát. Ilyen megoldással erősen támaszkodunk az öröklődésre és a dinamikus polimorfizmusra.

A Java első kiadásai ezt a megközelítést alkalmazták a szabványos tárolók megvalósításához. Jávában minden típus az `Object` típusból származik, azaz Jávában „minden `Object`” [4]. Így kézenfekvő volt az a megoldás, hogy az `ArrayList`, a `List`, a `Set` és a `Map`-hez hasonló konténerek `Object` típusú elemeket (pontosabban referenciákat) tárolnak. Ennek egyértelmű hátránya volt, hogy a tárolók úgymond nem „tudtak” a tárolt elemek típusáról semmit, így nem igazán lehet a konténer méretének lekérdezésén és az elemek (`Object` típusú referenciájának) lekérdezésén kívül mást csinálni. Ez könnyen megoldható egy-egy új „típus-tudatos” osztály bevezetésével:

```
public class ShapeList {
  private List list = new ArrayList();
  public void add(Shape sh) { list.add(sh); }
  public Shape get(int index) {
    return (Shape)list.get(index); // konverzió
  }
  public int size() { return list.size(); }
}
```

Így már „típus-tudatos” a `Shape` listánk, de ezt minden típus konténer párra ezt végig kellene csinálni, és lényegében ugyan azt a kódot írnánk le,

csak például `ArrayList` helyett `Set` szerepelne. Ez egy elég unalmas monoton munka, amibe az ember gyorsan belefárad és hibázik, viszont nagyon úgy tűnik, hogy egy számítógép könnyel el tudná végezni ezt a feladatot. És még nem is említettük meg a futás idejű pazarlást ami a `get()` hívásakor végrehajtott dinamikus típuskonverzió eredményez és az alakzatok virtuális függvény hívásainak költségét.

A C++ támogatja a típusos tömböket, így C++-ban nem lenne szükség a `ShapeList` osztályra, elég lenne egy `Shape* shapes[];` változó. Gyorsan belátnánk, hogy az alakzatokat, tömbben tárolni, ami egy összefüggő tárterület a memóriában, nem lesz hosszútávon túl szerencsés megoldás. Az új alakzatok hozzáadásával szükség lesz a tömb dinamikus bővítésére, vagyis jobb lenne ha egy könnyen és hatékonyan bővíthető listában tárolnánk az alakzatokat. Viszont szigorúan a C++ nyelvben, nincs beleépítve a láncolt lista, így azt vagy magunk implementáljuk, vagy egy előre megírt könyvtár segítségéhez fordulunk. Sajnos a második út a C++-ban, a sablonok bevezetése előtt nem igazán volt járható, vagyis, visszatértünk az előző `ShapeList` problémához, ugyanis nagy valószínűséggel a lista elemeit `void*` típussal deklarálták a könyvtár írói, hogy tetszőleges típusú elemet tudjon tárolni és újra nem tud a lista semmit a tárolt elem típusáról. Tehát akár melyik utat választjuk, akár saját lista implementációt, akár egy előre megírt `void*`-okat tároló listát választunk, nem kerülhetjük ki a lista típusossá tételéhez szükséges extra programozás, ami egyúttal hibaforrás is. Legalábbis ez volt a helyzet a sablonok bevezetése előtt.

1.2. Az első generikus C++ könyvtár

A konténereket és a rajtuk végrehajtandó algoritmusokat C++-ban nagyon elegáns módon oldották meg, osztály-sablonok és függvények-sablonok segítségével. A C++ STL könyvtárában (Standard Template Library, ami ma már része a C++ szabványnak), a konténerek (`vector`, `list`, `set` stb.) mind osztály-sablonok, melyeket a tárolandó elemek típusával paraméterezünk. Ezek mellé, az STL még a megfelelő algoritmusokat is biztosítja, mint például a `for_each`, a `sort` vagy a `find`. Az algoritmusok függvény sablonként vannak implementálva, és iterátorokon keresztül „kommunikálnak” a konténerekkel. Ez volt az első, és sokáig az egyetlen, hasznos generikus könyvtár implementáció.

Az STL megközelítésének számos előnye van az `Object`eket tároló konténerekkel szemben:

- **Típus biztonságos**: A fordító mindent elvégez helyettünk. Fordításiidőben példányosítja a osztály-sablonokat és függvény-sablonokat, és

a megfelelő típus ellenőrzéseket is elvégzi. Tehát a hibalehetőségeket drasztikusan lecsökkenti.

- **Hatékony**: Elkerüli a fölösleges, futás-idejű, dinamikus típus konverziókat. A sablonok segítségével a fordító mindent a fordítási-időben kiszámol, leellenőriz és behelyettesít – minden a fordításkor eldől semmit sem hagy futás-időre.
- **Egyszerűbb bővíteni és programozni**: akár új algoritmusokkal, akár új konténerekkel egyszerűen bővíthetjük a könyvtárat, csak a megfelelő iterátorokat kell implementálni.

Az utóbbiból az is adódik, hogy ha van m típusunk és n műveletünk, akkor az objektum orientál hozzáállással ellentétben, ahol minden típus minden (virtuális) metódusát, azaz $O(n*m)$ függvényt kell implementálni, generikus programozással viszont elég $O(n+m)$, vagyis külön-külön implementálnunk kell a típusokat (és iterátoraikat) és külön az algoritmusokat, egymástól teljesen függetlenül.

Ezek az előnyöket, az STL oly módon képes megvalósítani, hogy egyáltalán nem megy a programozó rovására, vagyis nem kell bonyolultabb kódot írni, sőt, kevesebb kódot kell írni, ami kevesebb hibával jár. A C++ sablonok, fordításkor manipulálják a kódot, és egy csomó rutin munkát elvégeznek a programozó helyett. Ebben az esetben, ahelyett, hogy mi írjuk meg különkülön az `IntVector`-t és a `FloatVectort`, a sablonok ezt automatikusan megteszik helyettünk, ugyanis lényegében ugyan azt a kódot kell megírni, csak más típussal.

Az STL megalkotásával le lettek fektetve a generikus programozás alapjai a C++-ban.

2. Sablon mágia – Template Metaprogramming

Ilyen sok problémát megoldani, ilyen hatékonyan szinte mágiának tűnhet. Sokáig annak is tűnt és ez arra ösztönzött pár C++ programozót, hogy kicsit jobban megvizsgálják a sablonok által nyújtott lehetőségeket. Így történt, hogy 1994-ben Erwin Unruh bemutatta a C++ szabvány bizottság előtt az első sablon-metaprogramot, amely, igaz nem fordult le, de viszont a hibaüzenetek prímszámok voltak. Felismervén a C++-ba beágyazott sablonmechanizmus, mint (fordítás idejű) nyelv Turing-teljességét, megfogant a C++ sablon-metaprogramozás (template metaprogramming) fogalma [3].

A sablonok, pontosabban a sablon-metaprogramozás, a funkcionális nyelvekhez hasonló módon működik. A részben specializált sablonok, függvény hívásnak felelnek meg és a teljes specializációk pedig termek, vagyis konstansok.

2.1. Fordítás-idejű aritmetika

Ezt nagyon jól szemlélteti a következő példa[9]:

```
// factorial.cpp
#include <iostream>

template <unisgned long N> struct Factorial
{ enum { value = N * Factorial<N-1>::value }; };

template <> struct Factorial<1>
{ enum { value = 1 }; };

// example use
int main()
{
    const int fact15 = Factorial<15>::value;
    std::cout << fact15 << endl;
    return 0;
}
```

Mi is fog történni amikor a fordító lefordítja a programunkat? A `fact15` egy $N = 15$-vel példányosított osztály-sablon, amelynek a `value` tagja egyenlő a `Factorial<N-1>::value`-vel, azaz a példányosítás „rekurzívan” történik, egész addig amíg a `Factorial<1>`-nél nem terminál. Nagyon fontos észrevenni, hogy ez mind fordítás időben történik, és futás időben a `fact15` változó, már a 1307674368000 literált kapja értékül. Vagyis nem írhatjuk a következőt:

```
int main()
{
   int fact; std::cin >> in;
   fact = Factorial<in>::value; \\ error: sablon nem
                                \\ példányosítható változóval
   std::cout << fact << endl;
   return 0;
}
```

ugyan is a fordító program nem találhatja ki, milyen számot fog a felhasználó bevinni.

Természetesen jogos az észrevétel, hogy minek írjunk egy furcsa, eleve nehezen érthető és talán félrevezető `Factorial` sablon-metaprogramot ha egy egyszerű kalkulátor programmal, vagy akár papíron is kiszámolhatjuk, hogy $15! = 1307674368000$. Így a következő program, ugyan azt tudja, csak sablon meta-programozás nélkül:

```
int main()
{
   int fact = 1307674368000; // = Factorial<15>::value helyett
   std::cout << fact << endl;
   return 0;
}
```

Hogy ezt a kérdést megvizsgáljunk, vegyünk talán egy másik gyakorlatiasabb példát példát, ugyanis a faktoriális csak egy iskolapélda a rekurzív függvény hívás lehetőségének bemutatására sablon meta-programozásban. A következő meta-program egy bináris alakban megadott szám értékét számolja ki:

```
template <unsigned long N>
struct binary
{
  static unsigned const value
    = binary<N/10>::value << 1 // prepend higher bits
    | N%10; // to lowest bit
};

template <> // specialization
struct binary<0> // terminates recursion
{
  static unsigned const value = 0;
};
```

Ezt a metaprogramot használva, a következő program a 42es számot fogja kiírni:

```
int main()
{
  std::cout << binary<101010>::value;
  return 0;
}
```

Talán ez a példa jobban szemlélteti a sablon meta-programozás lehetőségeit, ugyanis a `binary` segítségével, nem kell kézzel, vagy futás-időben számolgatni a bináris konstansokat, amire, ha például hardver közeli programot írunk, akkor lehet, hogy nem csak egyszer lesz szükségünk. Így nem is csak egy csomó számolást spóroltunk meg magunknak, hanem egy csomó hibát is kiszűrtünk. Ez lényegében a C++ kiterjesztésének is tekinthető, ugyanis a nyel már támogatta a hexadecimális és az oktális literálokat, most már a bináris literálokat is támogatja. Tehát

```
( 42 == 052 &&
  052== 2A &&
  2A == binary<101010>::value ) ; // true!
```

2.2. Típus aritmetika

Az elképzelhető, hogy a `binary` meta-programnak hasznát vehetjük valódi (pl. hardver közeli) programoknál, de azért a `Factorial` talán kicsit erőltetett „iskola" példa, ugyanis faktoriálist és mást numerikus aritmetikát valószínűbb, hogy futás-időben akarunk számolni. Viszont a sablon meta-programozás igazi ereje a típus aritmetikában rejlik.

Ez előző két példánál, `unsigned long` típusú paraméterrel kellet példányosítani a sablonokat, de jól tudjuk, hogy egy tetszőleges `T` típust is adhatunk át sablonoknak. A meghatározott típusú konstansokat és típus paramétereket, vagyis azokat a fogalmakat amit átadhatunk egy sablonnak *meta-adatnak* nevezzük.

A típus számolgatásokra egy szemléletes példa található D. Abrahams és A. Gurkovoy könyvében [3], ahol a sablon meta-programozással modellezik a fizikai mértékegységek kompatibilitását és olyan programozást tesznek lehetővé, hogy például egy hosszt és időt ábrázoló változó hányadosát csak egy sebességet ábrázoló változónak adhatunk értékül. Az említett példa a következő kódot teszi lehetővé:

```
quantity<float,length> l( 1.0f );
quantity<float,mass> m( 2.0f );

m = l; // fordítás idejü hiba
l = l + quantity<float,lengt>( 3.4f ); // ok

quantity<float,acceleration> a(9.8f);
quantitiy<float,force> f;

f = m * a; // ok: erö = tömeg*táv/idöˆ2
```

```
quantity<float,mass> m2 = f/a; // m2=a/f nem fordulna le
```

Természetesen a `m = l;` jellegű szűrést nem túl nagy ördöngösség észrevenni egy fordítónak. Az Ada programozási nyelvben, `type` paranccsal deklarált `mass` és `length` típusú változók, explicit konverzió nélkül, nem adhatók értékül egymásnak. Az igazi nehézség abban rejlik, hogy a fordító, meg tudja azt állapítani, hogy ha a tömeg (kg) típusú `m` változót szorzunk a gyorsulással (m/s^2) típusú `a` változóval, akkor erőt ($kg \cdot m/s^2$) kapunk, vagy hogy erőt ($kg \cdot m/s^2$) osztva gyorsulással (m/s^2) akkor tömeget (kg) kapunk. Tehát a megfelelő meta-programozással, az utolsó sort biztos, hogy nem fogjuk elrontani, vagyis ha elrontjuk és `m2=f/a` helyet `m2=a/f`-et írunk, jelez a fordító és a programunk nem fordul le. Az Ada csak annyit tudd, hogy jelez, hogy a tömeg és a gyorsulás nem ugyan az, a mi meta-programunk meg azt is tudja, hogy mikor melyik két típus alkot egy harmadikat.

Egy ilyen meta-programot összehozni nem kis feladat, tehát nem biztos, hogy megéri, viszont a Boost [5] Template Metaprogramming könyvtárai (Boost *MPL Library* [6], Boost `static_assert` [8] és Boost *TypeTraits*[7]) segítségünkre lehetnek. A Boost egy csomó segéd metaprogramot, metafüggvényt, metafüggvény osztályt, magasabb-rendű metafüggvényt stb. a rendelkezésünkre bocsájt, melyek nagyságrendekkel egyszerűbbé teszik a metaprogramozást. Sőt, a Boost MPL sokkal általánosabb és programozási szempontból sokkal hasznosabb eszközöket nyújt, mint az előző, fizikai mértékegységeket összeegyeztető példánk, de a Boost-ról és a könyvtárairól részletesebben majd később.

3. GTL

Sokáig az STL volt az egyetlen a gyakorlatban hatékonyan alkalmazható generikus könyvtár. Az STL bemutatta a generikus programozás lehetőségeinek és erejének egy részét, de mivel nem volt más példa és összehasonlítási alap, nem igaza tapasztalhattuk meg a generikus programozás teljes erejét és korlátait sem. Többek között, ez volt az egyik oka a GTL kidolgozásának.

A GTL az STL mintájára irták, viszont most a konténereket alakzatok (`poly`, `regpoly`, `circle`) helyettesítik, algoritmusokat pedig transzformációs műveletek(`move`, `rotate`, `mirror`).

3.1. Alkalmazás és algoritmusok implementációja

Az két dimenziós alakzatokat a csúcsaikkal tudjuk leírni. A csúcsok pontok a síkban vagyis két dimenziós vektoroknak is felfoghatjuk őket. A GTL a

csúcsokat `vect<T>` típusú osztály-sablonokkal írja le, ahol a `T` egy `double`-val kompatibilis típus. A `vect<T>`-ra definiálva vannak a szokásos műveletek, mint például a konstruktor, értékadás, egyenlőség vizsgálat és kiíró operátor. Ezeken kívül definiálva van még az összeadása, a kivonása két `vect<T>` között, `T`-vel (skalárral) való szorzás és az origó körüli elforgatás `double r` radiánnal.

3.1.1. Alakzatok

A GTL három alakzatot definiál:

1. Kör – `circle<T>`
2. Sokszög – `poly<T>`
3. Szabályos sokszög – `regpoly<T>`

Vizsgáljuk meg a kör implementációját részletesebben:

```
template<typename T>
class circle
{
public:
  circle( const vect<T>& origo=0, const T sugar=0 )
    : o(origo), r(sugar) {}

  vect<T> o; // origo
  T r;       // sugar
  // ...
```

A kört a `vect<T> o;` csúccsal és a `T r;` sugárral van reprezentálva. A konstruktor a megfelelő módon inicializálja ezeket a változókat. Ha paraméter nélkül hívjuk meg a konstruktort, akkor egy 0 sugarú kört kapunk az origón. Az iterátorokat ugorjuk egyenlőre át. Ez után következik pár segéd függvény:

```
template<typename T> class circle
{
public:
  // ...
  class input_iterator  { /* ... */ };
  class output_iterator { /* ... */ };

  void set( vect<T>& origo, T sugar )
```

```
  { o = origo; r = sugar; }

  input_iterator get_input_iter( int n=3 )
  { return input_iterator( o, r, n ); }

  output_iterator get_output_iter()
  { return output_iterator(*this); }

  input_iterator null_input_iter()
  { return input_iterator(); }
};
```

A `set()` metódussal tudjuk a sugarat és az origót közvetlenül módosítani, továbbá a többi alakzattól eltérően a kör input iterátor lekérdező metódusa paraméterezhető, ahol a paraméter egy pozitív egész, és az iterátor finomságát adja meg. Az iterátor finomsága alatt az kell érteni, hogy hány lépésben iterál végig a körön. Az alap értelmezett értéke ennek a paraméternek 3. Ekkor lényegében egy szabályos háromszög csúcsain lépked végig az iterátor, de ez bőven elég egy kör pontos meghatározásához.

A szabályos sokszög implementációja se túl bonyolult (eltekintve az iterátorok implementációjától):

```
template<typename T> class reg_poly
{
public:
  vect<T> o; // origo
  vect<T> p; // egy csucs
  int n;     // csucsok szama

  reg_poly(
    const vect<T>& origo=0,
    const vect<T>& csucs=0,
    int ncsucs=0 )
  : o(origo), p(csucs), n(ncsucs) {}

  // I/O iterátorok

  void set( vect<T>& origo,
    vect<T>& csucs, int ncsucs )
  { o = origo; p = csucs; n = ncsucs; }

  int size() const { return n; }
```

```
  // iterátor lekérdezések
};
```

A szabályos sokszöget szinte ugyan úgy reprezentáljuk mint a kört. A középpont most is a `vect<T> o;`. Mivel nem kerek az alakzat, ezért nem elég csak a sugarat megadni, hanem az egyik csúcsot is meg kell adni, ez most a `vect<T> p;`. Továbbá még a sokszög csúcsainak a számát is el kel tárolni az `int n;` változóban. A konstruktor és a `set()` hasonló mint a körnél. A szabályos sokszögnél értelmes a `size()` függvény is, ami a csúcsok számát adja vissza, hasonlóan ahogy az STL konténerek az elemek számát. Az iterátor lekérdezések is hasonlóak mint a körnél.

A (szabálytalan) sokszög kissé máskép van implementálva.

```
template<typename T> class poly
{
public:
  std::vector< vect<T> > v; // maguk a csucsok

  poly() {}

  // iterátorok és
  // iterátor lekérdezések

  void add( vect<T>& csucs )
  {
    v.push_back( csucs );
  }

  void del()
  {
    if( size()>0 ) v.pop_back();
  }

  int size()
  {
    return v.size();
  }

};
```

A sokszöget a `vector< vect<T> > v;` változó reprezentálja, amelyben a sokszög csúcsait tároljuk. Csak alapértelmezett konstruktor van definiálva,

az `add()` metódussal bővíthetjük a sokszöget egy új csúccsal, míg a `del()` metódus törli az utolsó csúcsot. A `size()` most is a csúcsok számát adja meg.

3.1.2. Iterátorok

Ahogy az STL könyvtárban már tapasztaltuk, az iterátorok kulcsfontosságúak lesznek. Mindegyik alakzatban, egy-egy alosztályként definiálva vannak a

```
class input_iterator{ /* ... */ };
class output_iterator{ /* ... */ };
```

iterátorok. Az elnevezés lehet, hogy nem épp a legszerencsésebb, ugyanis nem az alakzatok szempontjából, hanem az algoritmusok szempontjából nézve kapták ezeket a neveket. Az input iterátor az amelyik az algoritmusok inputját biztosítja, vagyis a „konstans" iterátor, ami az alakzat pillanatnyi állapotát tükrözi. Az output iterátor pedig az algoritmusok outputját biztosítják, vagyis amikor ezeket ezekre alkalmazzuk a `*` operátort akkor egy `vect<T>&` referencia szerű objektumot adnak vissza, amit módosítható.

Érdemes megfigyelni, hogy mind három GTL alakzat, az STL konténereivel szemben, nem lineáris hanem ciklikus. Ezért az iterátorok nem egy $[a, b)$ jellegű intervallumot járnak be, hanem körbejárják az elem csúcsait, és logikus, hogy az utolsó csúcs után, újra az elsőre lépnek. Ebből adódóan az alakzatok iterátorait célszerűbb ciklikus iterátoroknak, vagy cirkulátornak (circulator [10]) nevezni.

Mivel nem lineáris iterátorokról van szó, hanem ciklikusakról, nincs külön eleje és vége az intervallumoknak amit bejárnak, hanem a `begin()` és az `end()` megegyezik. Ez miatt elég csak egy-egy metódus amelyik az input és output iterátort tudja lekérdezni. Ezek a következők:

```
input_iterator get_input_iter();
output_iterator get_output_iter();
```

Mind három alakzatnak az iterátorai közös interfésze megtalálható a 3.1.2 és 3.1.2 táblázatokban. Ezen kívül minden alakzat input iterátora rendelkezik három konstruktorral:

```
input_iterator( ... ); // alakzattól függö paraméterek
input_iterator(void);  // a null_iterator() adja vissza
```

Ezen kívül az input iterátorok biztosítanak még egy-egy

```
null_iterator() { return input_iterator(); }
```

input_iterator		output_iterator	
név	érték/típus	név	érték/típus
iterator	input_iterator	iterator	output_iterator
value_type	vect<T>	value_type	vect<T>
reference_type	vect<T>&	reference_type	vect<T>&
pointer_type	vect<T>*	pointer_type	vect<T>*
		const_reference_type	const vect<T>&

1. táblázat. Input és output iterátor typedef-ek

input_iterator	output_iterator
bool operator== (const iterator&)const	bool operator== (const iterator&)const
bool operator!= (const iterator&)const	bool operator!= (const iterator&)const
value_type operator*()	iterator& operator*()
iterator& operator++()	iterator& operator++()
iterator operator++(int)	iterator& operator++(int)
iterator& operator--()	iterator& operator= (const_reference_type)
iterator operator--(int)	

2. táblázat. Input és output iterátor operátorok

metódust is. A paraméter nélküli konstruktor egy érvénytelen objektum iterátorát hoz létre, azaza a `null_iterator` egy ilyen iterátort ad vissza. Érvénytelen objektum alatt egy, valamilyen értelemben, null-objektumot értünk, mint például egy nulla sugarú kör vagy egy csúcs nélküli sokszög. Az első (paraméteres) konstruktor, a megfelelő alakzat csúcsain fog végig iterálni az `operator++` és `operator--` műveletekkel és az `operator*` operatorral lehet a csúcsot ábrázoló `vect<T>` vektort lekérdezni. Az alakzat `get_input_iter()` metódusa, az iterátornak amit létre hoz, át adja az összes szükséges adatott az alakzat leírásához. A körnél még a konstruktor paramétereként át lehet adni az iterátor „finomságát” is és ezek az iterátor adattagjaiban tárolódnak.

3.1.3. Algoritmusok

A GTL alakzataival végrehajtott algoritmusokat, a szerint, hogy módosítják-e az alakzatot, három csoportra oszthatjuk:

1. kizárólag „olvasó” algoritmusokra – az `input_iterator`t használják
2. kizárolag „író” algoritmusok – az `output_iterator`ral módosítják az

alakzatot

3. módosító algoritmusok – „író' és „olvasó" algoritmusok egyszerre

A Ezek közül az első és a harmadik csoport a hasznosabb és a GTL ilyen jellegű algoritmusokat valósít meg. A második csoport azért nem igazán alkalmazható, mert az `output_iterator`ok lényegében nem olvashatóak, és olyan algoritmusok tartoznak ide, mint például egy `T[]` típusú tömb vagy`vect<T>` vektorból előállít egy alakzatot. Ezeknek az algoritmusok már nem a GTL konténereivel dolgoznának, így túlmutatnak egy könyvtár hatáskörén.

Példa kizárólag olvasó algoritmusra a GTL `for_each` algoritmusa:

```
template<
  typename input_iterator,
  typename Functor>
void for_each( input_iterator mit, Functor func )
  {
    input_iterator save = mit;
    func(*mit++);
    while(mit!=save)
    func(*mit++);
  }
```

A `for_each` algoritmus nagyon jól szemlélteti, hogyan kell írni egy algoritmust a GTL-hez: elmentjük az iterátort (pl. `save` változóba) és előre olvasással addig iterálunk amíg körbe nem érünk (vagyis `mit!=save`). Egy függvény hívásnál nem szükséges talán, de ha nagyobb a `while` ciklus törzse akkor a `do...while()` konstrukció talán alkalmasabb az algoritmus(ok) előre-olvasó jellegéhez. Tehát talán a következő implementáció talán kissé jobb:

```
void for_each( input_iterator mit, Functor func )
  {
    input_iterator save = mit;
    do { func(*mit++); }
    while(mit!=save);
  }
```

Egy csomó ehhez hasonló, nem módosító, algoritmus található a GTL könyvtárban:

`void for_each(input_iterator i,Functor func)` Az iterátor által bejárt alakzat összes csúcsára meghívja az `f` functort.

`input_iterator find( input_iterator mit, Predicate pred )` Arra a csúcsra mutató iterátort adja vissza, amelyre a `pred` igaza.

`int count(input_iterator it, Predicate pred)` Megszámolja, hogy az iterátor által bejárt csúcsok közül hányra teljesül `pred`.

`bool isomorph(input_iterator1 i1, input_iterator2 i2)` Két alakzat egyenlő-e.

`value_type weight_point( input_iterator mit )` Kiszámolja az alakzat súlypontját.

Módosító algoritmusok azok, amelyek párhuzamosan léptetnek egy input és egy output iterátort és az egyiket olvassák és ennek alapján a másikat módosítják. A GTL a következő módosító algoritmusokat biztosítja:

`void move( ..., const_reference_type v )` Az alakzat eltolása `v` vektorral.

`void mirror( ..., const_reference_type v )` A `v` ponton áthaladó függőleges tengelyre való tükrözés.

`void rotate( ..., double rad )` At alakzat elforgatása az origó körül `rad` fokkal.

`void inplace_rotate( ..., double rad )` At alakzat elforgatása a súlypontja körül, `rad` fokkal.

`void clone( ... )` At alakzat klónozása/másolása. [1]

[1] A `clone` egy speciális eset, ugyanis ezt nem *egy* alakzat input és output iterátorára érdemes alkalmazni, hanem ha egy alakzatot egyenlővé akarunk tenni a másikkal.

Hivatkozások

[1] Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu, 2001, [1305], ISBN 963 9301 17 5 ö

[2] Zoltán Porkoláb, Róbert Kisteleki: *Alternative Generic Libraries*, ICAI'99 4th International Conference on Applied Informatics, Ed: Emőd Kovács et al., Eger-Noszvaj, 1999, [79-86]

[3] David Abrahams, Aleksey Gurtovoy: *C++ Template Metaprogramming - Consepts, Tools, and Techniques from Boost and Beyond*, Addison Wesley Professional, 2004, ISBN 0-321-22725-5

[4] Bruce Eckel, *Thinking in Java (3rd Edition)* , Prentice Hall PTR, 2002, [1119], ISBN-10: 0131002872

[5] *Boost C++ Libraries*
`http://www.boost.org/libs/libraries.htm`
2007. május 30.

[6] *The Boost MPL Library*
`http://www.boost.org/libs/mpl/doc/index.html`
2007. május 30.

[7] *Boost.TypeTraits*
`http://www.boost.org/doc/html/boost_typetraits.html`
2007. május 30.

[8] *Boost.StaticAssert*
`http://www.boost.org/doc/html/boost_staticassert.html`
2007. május 30.

[9] Dr. Zoltán Porkoláb: *Advanced C++ Lessons*
`http://aszt.inf.elte.hu/~gsd/halado_cpp/`
2007. május 30.

[10] *Computational Geometry Algorithm Library – CGAL*
`http://www.cgal.org/`
2007. május 30.

[11] *View Template Library – VTL*
`http://www.zib.de/weiser/vtl/`
2007. május 30.

[12] Vatai Emil: *GTL Draw – Nagyprogram fejlesztői dokumentáció*, ELTE-IK, Programtervező Matematikus szak, Bemutatott nagyprogram dokumentáció, [19]