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 11.

Tartalomjegyzék

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 eszközeit kellett 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) mellett nem kerülhetőek meg.

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

1. A Generikus programozás kialakulása

A C++ első kereskedelmi forgalomba hozatalakor, 1985-ben még nem támogatta 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 az 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, melyet egy `DrawAll` függvénnyel rajzolunk ki, amely a tömb minden elemére meghívja 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]. 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 és az elemek (`Object` típusú referenciájának) lekérdezésén kívül mást tenni. 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 minden típus-konténer párra ezt végig kellene csinálni, és lényegében ugyan azt a kódot írnánk le, csak

például `ArrayList` helyett `Set` szerepelne és a `Shape` helyet egy másik típus. Ez egy elég unalmas monoton munka, amibe az ember gyorsan belefárad, tehát könnyen hibázhat, viszont nagyon úgy tűnik, hogy egy számítógép könnyen el tudná végezni ezt a feladatot.

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 programozást, ami egyúttal hibaforrás is lehet, 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 paraméterezhetünk. 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. Sokáig az egyetlen, hatékonyan használható 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- és függvény-sablonokat, a megfelelő típus ellenőrzéseket elvégezve, ebből eredően a hibalehetőségeket drasztikusan lecsökkentve.

- **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.

- **Egyszerű bővíteni és programozni**: Akár új algoritmusokkal, akár új konténerekkel egyszerűen bővíthetjük a könyvtárat, csak a megfelelő függvény-sablonokat vagy osztályokat és iterátorokat kell implementálni.

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

Ezeket az előnyöket, az STL oly módon képes megvalósítani, hogy egyáltalán nem megy a programozó rovására, vagyis nem kell bonyolultabb kódot írni, sőt kevesebbet, ami kevesebb hibát eredményez. 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 `IntVectort` 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 varázslásnak tűnhet. Sokáig annak is tűnt és ez arra ösztönzött néhány 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, 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.

A sablonok, pontosabban a sablon-metaprogramozás, a funkcionális nyelvekhez hasonló módon működik. A részben specializált sablonok, függvény

hívásoknak felelnek meg és a teljes specializációk pedig termek, vagyis konstansok.

2.1. Fordítás-idejű numerikus aritmetika

Vegyük szemügyre a következő példát[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 }; };

// alkalmazva
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` egyenlő egy $N = 15$-vel példányosított osztály-sablon `value` tagjával, ami N szer `Factorial<N-1>::value`-vel egyenő, 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. Tehát a következő program, ugyan azt teszi, csak sablon meta-programozás nélkül:

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

Erre a kérdésre, hogy válaszoljunk, vegyünk egy másik gyakorlatiasabb példát, ugyanis a faktoriális csak egy iskolapélda a rekurzív függvény hívás lehetőségének bemutatására sablon meta-programozásban. A következő metaprogram 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. Ezzel nem csak egy csomó számolást spóroltunk meg magunknak, hanem a lehetséges hibákat is kiszűrtük. Ez lényegében a C++ kiterjesztésének is tekinthető, ugyanis a nyelv már támogatta a hexadecimális és az oktális literálokat, most már a bináris literálokat is támogatja. Tehát a következő egyenlőségek, mind igazak:

```
( 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 más 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 kellett 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 távolságot 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 az `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 gyorsulás (m/s^2) típusú `a` változóval, akkor erőt ($kg \cdot m/s^2$) kapunk, vagy ha erőt ($kg \cdot m/s^2$) osztunk 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 tud, hogy jelzi, hogy a tömeg és a gyorsulás nem ugyan az, a mi meta-programunk pedig 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. bocsájt a rendelkezésünkre, melyek nagyságrendekkel egyszerűbbé teszik a metaprogramozást. Sőt, a Boost MPL sokkal általánosabb és programozás szempontból sokkal hasznosabb eszközöket is nyújt, mint az előző, fizikai mértékegységeket összeegyeztető példa, de a Boost-ról és 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 igazán tapasztalhattuk meg a generikus programozás teljes erejét és korlátait sem. Többek között, ez volt az egyik oka a GTL kifejlesztésének.

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

A 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>`-re definiálva vannak a szokásos műveletek, mint például a konstruktor, értékadás, egyenlőség vizsgálat és kiíró operátor.

Ezeken kívül definiálva van még az összeadás, a kivonás két `vect<T>` között, `T`-vel (skalárral) való szorzás és az origó körüli elforgatás `double r` radiánnal.

3.1. Alakzatok

A GTL három alakzatot definiál:

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

3.1.1. Kör – circle

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 kör 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 azt kell érteni, hogy hány lépésben iterál végig a körön. Az alap értelmezett értéke ennek a paraméternek 0, és ekkor az origót adja vissza.

3.1.2. Szabályos sokszög – reg_poly

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 hasonlóan reprezentáljuk mint a kört. A középpont most is a `vect<T> o;`. Mivel nem kerek az alakzat, ezért nem elég csak a sugarat, 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.

3.1.3. Sokszög – poly

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 vissza.

3.2. Iterátorok

Ahogy az STL könyvtárnál már tapasztaltuk, az iterátorok kulcsfontosságú szerepe töltenek be. Mindegyik alakzatban, egy-egy alosztályként definiálva vannak a

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

iterátorok. Az elnevezés lehet, hogy nem éppen a legszerencsésebb, ugyanis nem az alakzatok, 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 ezekre alkalmazzuk a `*` operátort akkor egy `vect<T>&` referencia szerű objektumot adnak vissza, ami 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 alakzat csúcsait, és logikus, hogy az utolsó csúcs után, újra az első következik. 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 1 és 2 táblázatokban. Ezen kívül minden alakzat input iterátora rendelkezik két konstruktorral:

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

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

Ezen kívül az input iterátorok biztosítanak még egy-egy

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

metódust is. A paraméter nélküli konstruktor egy érvénytelen objektum iterátorát hozza létre, azaz 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*` operátorral lehet a csúcsot ábrázoló `vect<T>` vektort lekérdezni. Az alakzatok input iterátor lekérdező `get_input_iter()` metódusa, az iterátornak melyet 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.

3.3. Algoritmusok

A GTL alakzataival végrehajtott algoritmusokat, aszerint, 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.

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 csak olyan algoritmusok tartoznának ide, mint például egy `vect<T>[]` típusú tömbből egy alakzat előállítása. Ezek az algoritmusok már nem a GTL konténereivel dolgoznának, így túlmutatnak eme könyvtár hatáskörén.

3.3.1. Nemmódisító algoritmusok

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`). Egyetlen függvény hívásnál nem szükséges, de ha nagyobb a `while` ciklus törzse akkor a `do...while()` konstrukció talán jobban illeszkedik az algoritmusok előre-olvasó jellegéhez. Tehát 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[1] iterátort adja vissza, amelyre a `pred` igaz.

`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.

Ezek az algoritmusok csak az alakzat input iterátorát használják. Tipikussan a következő módon alkalmazhatjuk őket (a példában egy sokszög, a koordináta rendszer első kvadránsába eső csúcsait a számoljuk):

```
bool quad1( vect<float> v )
{ return ( v.x >= 0 ) && ( v.y >= 0 ); }

void f( reg_poly<float> rp )
{
  int cnt;
  cnt = count( rp.get_input_iter(), quad1 );
  std::cout << "csúcsok száma: " << cnt << std::endl;
}
```

3.3.2. Módisító algoritmusok

Módosító algoritmusok azok, amelyek párhuzamosan léptetnek egy input és egy output iterátort. 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.

[1] A kifejezés arra utal, hogy ha az iterátorra alkalmazzuk a dereferencia operátort, akkor pont típusú objektumot ad vissza. Ez a szóhasználat később is megtalálható a dolgozatban és az iterátorok imént említett viselkedésére utal.

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

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

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

`void clone( ... )` Az alakzat klónozása/másolása. [2]

A felsorolt függvény-sablonok paraméter listája nem teljes, ugyanis a `...` helyén, az `input_iterator mit, output_iterator hova` áll. Például a `move` teljes függvény-sablon prototípusa:

```
template<
  typename input_iterator,
  typename output_iterator,
  typename T>
void move(
  input_iterator mit,
  output_iterator hova,
  const vect<T>& v )
```

A módosító algoritmusokra talán a `move` a legegyszerűbb és legszemléletesebb példa:

```
void move(
  input_iterator mit,
  output_iterator hova,
  const vect<T>& v )
{
  input_iterator save = mit;
  do{ *hova++ = (*mit++) + v; }
  while( mit!=save )
}
```

Ezeket a műveleteket tipikusan a következő módon lehet alkalmazni (a példában a `move` műveletet alkalmazzuk egy körre):

[2] 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.

```
void f( circle<float> c, vec<float> v )
{
  move( c.get_input_iter(), c.get_output_iter(), v );
}
```

Az eddig felsorolt, alap algoritmusok mintájára, nagyon könnyen lehet tetszőleges algoritmust implementálni. Láthatjuk, hogy az STL-hez hasonlóan, csak az iterátorokat, és az algoritmushoz szükséges információt (functort vagy elforgatás szögét), kell átadni.

4. Alkalmazás – GTL Draw

A GTL és ezzel a generikus programozás lehetőségeinek és korlátainak feltárása céljából készült a *GTL Draw* rajzoló program (egy korábbi munkám ami nagyprogramként lett bemutatva). A program a következő funkciókat valósítja meg:

Alakzatok rajzolása Kör, sokszög és szabályos sokszögek rajzolása.

Alakzatok módosítása A már megrajzolt alakzatok módosítása, vagyis az alakzatok eltolása, forgatása és tükrözése.

4.1. Az implementáció lényegesebb pontjai

A GTL és a generikus programozás vizsgálat érdekében, a GTL Draw implementációjának következő pontjait érdemes megemlíteni:

1. `double` lebegőpontos típussal van példányosítva mind három GTL alakzat. Ezért a `double`ra be van vezetve `typedef`fel egy `scalar` szinonima. További szinonimák vannak bevezetve a példányosított alakzatokra és a vektorokra amelyekben tárolódnak:

   ```
   typedef double scalar;
   typedef gtl::vect<scalar> Vect;

   typedef gtl::circle<scalar>Circle;
   typedef gtl::poly<scalar>Poly;
   typedef gtl::reg_poly<scalar>RegPoly;

   typedef std::vector<Circle>vCircle;
   typedef std::vector<Poly>vPoly;
   typedef std::vector<RegPoly>vRegPoly;
   ```

2. A rajzlap tartalmát három vektor tárolja. Sorra `vCircle`, `vPoly` és `vRegPoly` típusú `m_vCircle`, `m_vPoly` és `m_vRegPoly` vektor tárolja a megfelelő alakzatot.

3. A program hat lényegesebb állapotban lehet: három állapot jellemzi a három alakzat rajzolását, három másik állapot pedig a három transzformáció végrehajtását.

4. A program lekezel minden eseményt. Ezek közül a kattintás a rajzlapra a legösszetettebb. Az állapot függvényében, a következő történik:

   - Ha alakzatot rajzolunk, akkor a megfelelő vektor bővül egy alakzattal.
   - Ha transzformációs műveletet hajtunk végre, akkor először végigkeresi a program mind három vektort, hogy volt-e találat (alakzatra kattintottunk-e). Ha volt találat akkor az alakzat címét, amelyre rá lett kattintva, a `Circle* pc`, `Poly* pp` és `RegPoly* rp` mutató közül, a megfelelőben tárolja és a másik kettőt nullára állítja. Majd mind a három mutatóra (pontosabban a mutató által mutatott alakzatra) meghívódik a megfelelő algoritmus, feltéve, hogy a mutató nem nulla.

5. Amikor szükség volt rá, három `Draw` sablon-függvény hívással kirajzolódtak a három vektorban tárolt alakzatok.

A program implementációjának részletes leírása megtalálható az [12] címen, a *doc* könyvtár alatt.

Fontos megszorítás, hogy a GTL Draw program megírásakor, a generikus programozás tanulmányozása érdekében, abszolút kikötés volt, kizárólagosan sablonok, illetve sablon könyvtár használata. Azt tapasztaltam, a program írása során, hogy a (szigorúan) generikus programozás, ezen megszorítások és kikötések mellet, nem teljesen alkalmas egy ilyen rajzprogram megírásához. A generikus könyvtár nagyon nagy erővel bír és óriási lehetőség rejlik benne, de dinamikus polimorfizmus nélkül komoly gondot okoz a különböző típusú alakzatok (konténerek) tárolása, ugyanis nem lehet őket egy közös konténerbe tenni. Megpróbálhatjuk elemezni a GTL Draw-ban alkalmazott generikus programozás előnyeit, de ez elég ambivalens és vitatható eredményekhez vezet.

4.2. Előnyök

A generikus programozás egy olyan programozási paradigma, mely metaprogramozásra épül, azaz fordítás idejű programozásra. Egy jól megírt meta-

program vagy generikus sablon könyvtár, kiszámolja és el intézi a programozó helyett azt, ami eldönthető és kiszámítható fordítási-időben. A generikus programozásnak általános előnyei:

- Típus biztonság
- Hatékonyság
- Egyszerű kód

4.2.1. Típus biztonság

A típus biztonság egyértelműen teljesül. Sehol sem nincs a programban semmiféle típuskonverzió, minden objektumnak jól definiált típusa van. Ezekkel a típusokkal megfelelően példányosítódnak a megfelelő függvény sablonok. Ha nem jó típusú objektumot adunk át, akkor azt a fordító rögtön jelzi, és a program nem fordul le.

Ez alól egy apró kivétel a mutatók használata, de azok csak lokális változóként szerepeltek és könnyű belegondolni hogyan lehet megkerülni a mutatók használatát. Viszont még így is csak az okozhatott futás-idejű hibát, ha a null mutatóra hívjuk meg az algoritmust, de ezt könnyen ki lehet kerülni.

4.2.2. Hatékonyság

A hatékonyság abból adódik, hogy teljes mértékben el vannak kerülve az öröklődés, a virtuális függvények és a típus konverziók. A fordító program, például az alakzatok kirajzolásakor, fordítás-időben létrehozza a megfelelő kódot, ami futás időben lefut.

Attól függetlenül, hogy nincsenek „rejtett” konverziók és „rejtett” virtuális függvények (azaz függvény hívások) a programban, és a program pontosan azt fogja csinálni ami le van kódolva, csak „bővebben kifejtve”, látszik, hogy pár dolog talán ront egy picit a teljesítményen. Itt újra fel lehet hozni a mutatókat, és a null mutató vizsgálatát.

4.2.3. Egyszerűbb kód

Ez a legvitathatóbb „előny”, ugyanis az *egyetlen* rajzlap tartalmát *három* különböző vektorban kell tárolni, nem igazán mondható elegáns és egyszerű megoldásnak. Ez annak a következménye, hogy teljes mértékben mellőztük az öröklődést és nincs egy ősosztályunk, ami tudna kör, sokszög és szabályos sokszög is lenni, és ilyen típusú vektorban vagy listában tárolhatnánk a rajzlap tartalmát. Ezért minden műveletet, mint például a kirajzolás, a „találat keresés” (annak az alakzatnak a keresése, amelyre rákattintottunk),

háromszor kell végrehajtani, különböző paraméterrel, a mutatós komplikációkat nem is említve.

Azt viszont meg kell jegyezni igaz, hogy háromszor kellet meghívni például a `Draw` függvényt, de csak egyszer kellet megírni és mindhárom alakzatra működött. Pontosabban kettő `Draw` függvény lett implementálva, ugyanis a kört, egy specializált `Draw` függvény sablon rajzolta ki. Ezt is talán hatékonyabb kódolásnak lehet tekinteni, mivel lehetőséget nyújtott, a megjelenítő keretrendszer kör-rajzoló függvény alkalmazására, de ugyanakkor, az általános `Draw` függvény sablon is ki rajzolta volna a kört[3], csak kevésbé hatékonyan.

4.3. Hátrányok

Ahogy már korábban is említettük, a fő hátrány az, hogy a sablonok által generált osztályok teljesen függetlenek egymástól. Nincs közös ősosztályuk, és ezért nem lehet egyszerre, úgymond „egy kalap alatt" kezelni őket. Hiányzik az a lehetőség, hogy például egy `list<Shape*>` listában tároljuk őket. Ez ahhoz vezet, hogy a három különböző típushoz, három különböző változóra van szükség.

A program írásának legkényelmetlenebb része volt a kiválasztott alakzat elmentése, ugyanis nem volt elég csak azt elmenteni, hányadik alakzat a tömbben, hanem azt is ki kell valahogyan találni, hogy melyik tömbben. Ezt persze csak úgy lehet, hogy mind a három tömböt végig kell nézni előtte, miután a három mutató közül kell kiválasztani azt, hogy melyik az érvényes.

4.4. A polimorfizmus megkerülhetetlensége

Az a kérdés merülhet fel bennünk, sőt fel is kell, hogy merüljön: *Meg lehet-e ezt a problémát oldani generikus programozással, vagyis sablon meta-programozással?* A válasz erre egyáltalán nem triviális és azt fogjuk belátni, hogy a válasz erre a kérdésre nemleges.

4.4.1. Generikus programozás egy definíciója

David R. Musser, a generikus programozás egyik úttörője, a honlapján azt írja a generikus programozás definíciójáról, hogy az: „programming with concepts", azaz programozás conceptekkel. Az angol concept, illetve a conception szó jelentése magyarul: fogalom, elgondolás, elképzelés, eszme, felfogás. Viszont a concept szónak egy új értelmezése is született a közelmúltban.

[3]Az iterátor finomságán tudunk állítani, ha az input iterátor lekérdező függvényt egy paraméterrel látjuk el. Elég nagy finomság mellet, elvileg azonos eredményt érünk el.

Concept fogalmát nemrég vezeték be a C++-ba. Még hivatalosan a concept ellenőrzés nem része a szabványnak de hamarosan az lesz.

A concept a sablonoknál olyasmi, mint az interfész az öröklődésnél. Ha adott egy sablon, melynek van egy nem specializált `T` típus paramétre, akkor concept-ekkel lehet meghatározni azt, hogy szemantikailag és szintaktikailag mit várunk el a `T` típus helyén példányosítandó típustól.

Például az iterátoroktól elvárjuk, hogy lehessen léptetni (`operator++`), lehessen az értékét lekérdezni (`operator*`) és lehessen egyenlőséget illetve egyenlőtlenséget vizsgálni (`operator==` és `operator!=`). Ezeknek a pontos meghatározása az algoritmusok alkalmazásánál nagyon hasznos lenne, ugyanis a C++ mai eszközei, nem teszik lehetővé, hogy egy nem általunk írt könyvtár generikus algoritmus paramétereiről bármit tudjunk. Ha egy egyszerű függvényt hívunk meg, a szignatúrája mindent elmond a paramétereiről, míg egy sablon deklaráció csak azt árulja el, hogy hány különböző paraméterrel lehet példányosítani, és hogy ezek közül melyik egy típusnév és melyik egy bizonyos meghatározott típusú értékek. A concept-ek specifikálnák, hogy egy típusnak, amellyel egy sablont szeretnénk példányosítani, milyen metódusokat, altípusokat és `typedef`eket kell tartalmazniuk.

A generikus programozás ereje, abban rejlik, hogy ha az adatszerkezeteinket „hasonlóan" írjuk meg, vagyis úgy, hogy megfeleljenek bizonyos concept-eknek, akkor könnyű ezekre az adatszerkezetekre olyan algoritmusokat írni, melyek függvény-sablonokként vannak implementálva. A generikus programozással megvalósított algoritmusok, szintaktikailag lényegében azonos kódot generálnak, melyek csak a paraméterként átadott típusban különböznek. Ebből adódóan, a fordító program, legenerálja nekünk azt a kódot amire nekünk szükségünk van, de a döntéseket nem halaszthatjuk el futás-időre. A sablonok kizárólag fordítási időben játszanak szerepet.

Ezért van az, hogy a generikus programozás fő alkalmazási területe a könyvtárak fejlesztése mivel ott a leghatékonyabb. Konkrét program írásánál, nem nagyon fogunk generikus programozást használni, ugyanis ott már inkább a generikus programozás gyümölcseit (a generikus könyvtárakat) fogjuk használni. Egy programozó nem fog csakis azért iterátorokat implementálni és meta-programokat írni, hogy egy konkrét feladathoz, egy konkrét problémát megoldjon. A generikus programozás csakis akkor kifizetődő stratégia, ha nagymértékben újrahasználható, hatékony kódot szeretnénk létrehozni. Talán úgy lehetne, a generikus programozásra gondolni, mint az objektum orientált programozás megfelelőjére, azzal a különbségekkel, hogy a generikus programozás könyvtárak fejlesztésére alkalmasabb mivel a fordításig halasztja a döntéseket, hogy a könyvtár felhasználja az alkalmazás írásakor dönthesse el hogyan valósítja meg a programot, míg az objektum orientált programozás alkalmazások fejlesztésére alkalmasabb mivel futás-időre ha-

lasztja el a döntéseket, hogy az alkalmazás felhasználja, futás-időben mondja meg, mit csináljon a program.

4.4.2. GTL kontra STL

Első ránézésre a GTL-t nem nehéz megírni az STL analógiájára, viszont ha jobban szemügyre vesszük, akkor lényeges különbségeket fedezhetünk fel a két könyvtár között. Az analógia szembetűnő és nagyon egyszerű:

- A GTL alakzatai, az STL konténereinek felelnek meg.
- A GTL algoritmusai az alakzatokra vannak megírva, míg az STL algoritmusai a konténerekre.

Viszont ha belenézünk a GTL forrásába, akkor az első szembetűnő dolog a `vect<T>` osztály sablon. Mivel síkbeli alakzatokat szeretnénk ábrázolni és az alakzatokat a jellegzetes pontjaikkal határozhatjuk meg (csúcsok, középpont), ezért szükség van a pont definíciójára, amit egy valamilyen számtest feletti 2 dimenziós[4] vektortér eleme ábrázol. Továbbá a `vect<T> rotate` metódusában megfigyelhető, hogy a `T` egy olyan numerikus típus kell, hogy legye, amely `double`ra konvertálható.

A GTL úgy van elképzelve, hogy az alakzatok olyan konténerek amelyek pontokat tárolnak. Tehát attól függetlenül, hogy paraméterezhető osztálysablonként vannak implementálva, a paraméter nem fogja meghatározni mit fog tárolni, ugyanis mindig pontokat tárol, sőt `double`ra konvertálható `T` típussal példányosított `vect<T>` vektorokat (vagyis autóbusz típussal nem valószínű, hogy sikerül példányosítani sem az alakzatokat, de még a vektorokat sem). Ez feltűnően elüt az STL sokoldalúságától és flexibilitásától, mivel az STL konténerei bármilyen típusú objektumokat tárolhatnak.

A szakértők szerint, az objektum orientált paradigmát akkor érdemes alkalmazni, amikor az objektumoknak hasonló a struktúrájuk, míg a generikus paradigmát akkor amikor hasonló a viselkedésük. Az alakzatokra egy kissé mindkettő igaz, vagyis egyik sem. Hasonló a struktúrájuk, mivel mindegyik alakzat, valamilyen módon pontokat tárol, viszont különbözik is a struktúrájuk, mivel a kör csak a középpontot és a sugarat tárolja tárolja, a szabályos sokszög a középpontot, az egyik csúcsot és a csúcsok számát, míg az általános sokszög az összes csúcsot egy vektorban tárolja, mégis mind három alakzat iterátora `vect<T>` típusú értékre mutat. A viselkedésük is hasonlónak vélhető, ugyanis ciklikusak, letudjuk kérdezni az egyik csúcsot, de ugyan akkor különbségeket is felfedezhetünk, mint például az, hogy a sokszögnek és a szabályos sokszögnek lekérdezhetjük a méretét (`size()`), míg a körnél ez a

[4]Természetesen nem lenne nehéz megvalósítani az n dimenziós, általános esetet.

művelet nem értelmes, ugyanígy a sokszöghöz adhatunk hozzá új csúcsokat és törölhetünk is belőle, ugyanakkor a másik kettő ezeket a műveleteket nem támogatja. Ez is azt indokolja, hogy az alakzatokat nem célszerű teljesen és szigorúan generikus programozással megvalósítani.

Ezzel szemben az STL az alap típusok azonos viselkedésére és a mutatók viselkedésére támaszkodik. Az STL olyan konténereket és algoritmusokat valósít meg, amelyek olyan alapvető műveleteket követelnek meg a példányosítandó típusoktól, mint például az alapértelmezett konstruktor, értékadás operátor, egyenlőség vizsgálat operátor, vagy az iterátoroknál az inkrementállás, a dereferenciálhatóság, két iterátor különbsége stb. Ezeket a beépített típusok és mutatók mind teljesítik, ahogy az STL adatszerkezetei és iterátorai is. Ebből adódik a fordítás-idejű polimorfizmus, és ezért viselkednek úgy az STL típusai mintha beépített típusok lennének, vagyis ezért gondolhatunk az STL-re mint a nyelv bővítésére. A GTL polimorfizmusát viszont inkább csak az biztosítja, hogy az iterátorok mindig egy `vect<T>`-re mutatnak.

Végül a legszembetűnőbb különbség az STL és a GTL között az, hogy melyik hol használja ki a polimorfizmust. Az STL abban jeleskedik, hogy egyik konténert kicserélhetjük a másikra, és egy újrafordítás után, nagy valószínűséggel minden tökéletesen fog működni. Viszont ez program tervezési döntés segítése, vagyis alkalmazása. Ha például egy vektor helyet egy listában szeretnénk tárolni az adatainkat, akkor ezt a program tervezéskor fogjuk megváltoztatni. Nem fogunk egy olyan programot írni, amely futás időben változtatja meg a belső reprezentációját, a GTL Draw-ban pedig pontosan ez történik: például az eltolásnál (vagy más transzformáció alkalmazásánál) attól függően, hogy milyen alakzatra kattintottunk, attól függően fog a `move` függvény-sablon példányosításai közül az egyik meghívódni.

Itt láthatunk még egy nagy különbséget a két könyvtár között. Míg egy, az STL könyvtárat használó, programban a fordító csak azokat az függvény-sablonokat (algoritmusokat) csak azokkal a osztály-sablonokkal (konténerekkel) példányosítja, amelyeket a programozó a tervezés folyamán hasznosnak talált. A GTL Drawban és valószínűleg ez más GTL-re épülő rajzoló programokra is igaz lenne, hogy a fordítónak példányosítania kell majd a tervezés során használt összes függvény-sablont (transzformációt) az összes osztály-sablonnal (alakzattal). Tehát ha a programunkban van egy listánk és egy vektorunk, és a listában keresni kell a vektort meg rendezni kell, akkor a `find` csak a listával példányosul, a `sort` meg csak a vektorral, míg nem valószínű, hogy egy rajzolóprogramot úgy szeretnénk megírni, hogy csak háromszögeket tudjon forgatni és csak poligonokat eltolni. Ezért a GTL Draw fordításako, minden függvény-sablont minden osztály-sablonnal példányosul még ha nem is fog lefutni a program futtatása kor.

4.4.3. Összetett alakzat

Egy alapvető hiányossága a GTL-nek az összetett alakzat (`composite`), ami lényegében megoldaná a fejlesztés alatt felmerülő összes problémát. Az összetett alakzat alatt egy olyan típust értünk, amelyik több fajta alakzatból áll és lehetővé teszi, hogy ezeken az alakzatokon végigiteráljunk. Első nekifutásban, azt gondolja az ember, hogy nem is olyan nagy probléma: létrehozunk egy `composite<T>` osztály-sablont, ami egy listában tárolja az alakzatokat. De mivel nincs közös bázisosztályunk, ezért nem tudjuk egy listába rakni a különböző típusú alakzatainkat.

Az összetett alakzat problémája körülbelül egyenértékű az eredeti problémánkkal, a polimorfizmus hiányával. Majd látni fogjuk, hogy kicsit finomítunk a problémán és közelebb jutunk egy kompromisszumos megoldáshoz.

Az alakzatok kirajzolása szempontjából teljesen mindegy, hogy a polimorfizmust sikerül-e megkerülni vagy egy összetett típust tudunk megvalósítani. Ha valahogy sikerülne megkerülni a polimorfizmust és sikerülne egy konténerbe helyezni az összes alakzatot, akkor egy ilyen konténerrel a `composite`ban is tudnánk tárolni az objektumokat. Másrészt ha létre tudunk hozni egy összetett alakzatot, amely tetszőleges alap alakzatokat tartalmaz, akkor egy nagy összetett alakzat típusú objektumban tudjuk tárolni az egész rajzlap tartalmát is, és azt kell kirajzolni.

Ha jobban belegondolunk, ez egyrészt nem valósítható meg valamilyen öröklődés nélkül vagy típuskonverzió nélkül (dinamikus polimorfizmus) másrészt ha még sikerülne is kikerülni a polimorfizmust és a típuskonvertálgatásokat, akkor is homály az, hogy milyen iterátorokkal fogunk dolgozni, de erről majd később.

Vizsgáljuk meg előtte a keresés (vagyis „találat” keresés, annak megállapítása, hogy melyik alakzatra kattintottunk) szempontjából a két problémát. Egyrészt a polimorfizmus szigorúbbnak tűnhet, ugyanis ha megengedjük a dinamikus polimorfizmust, például öröklődéssel, akkor tudunk csinálni alakzatra mutató mutatót és ebben tárolhatjuk azt az alakzatot a amelyre rákattintottunk. Viszont az ha csak összetett alakzatunk van, és például ebben tároljuk a rajzlap tartalmát, akkor az szigorított feltétel, hogy le tudjuk menteni, az összetett alakzat egyik alkotó elemét.

4.4.4. Iterátor konkatenáció

Már volt rá utalva, hogy lényegében nem is az alakzatokat kellene összefűzni hanem az iterátorokat. Az összetett alakzat kirajzolásánál is egy alapvető problémába ütközünk: hogyan fogjuk végig iterálni az alakzatokat. Egyrészt, hogyan fogjuk tudni, hogy egy alakzat végére értünk, másrészt milyen típusú

iterátorral fogunk iterálni, ugyanis a különböző konténerek iterátorai, különböző típusúak, meg ha teljesen azonos is a szemantikájuk. A teljesen azonos szemantika pedig teljesül: feltéve, hogy azonos `scalar` típussal példányosítottuk az alakzatainkat, akkor világos, hogy mindhárom alakzat iterátora egy-egy `vect<scalar>` típusú pontra mutat és ha léptetjük akkor körbejárja az alakzat csúcsait.

Újra felmerül a kérdés, hogy objektum orientált vagy generikus megoldást kellene alkalmazni. Az öröklődés újra kézenfekvő megoldás. Az iterátoroknak lenne egy közös bázis iterátor osztálya, virtuális metódusokkal, az alakzatok helyet az iterátorokat kellene csak tárolni, például az egyik tömbben az input iterátorokat, a másik tömbben az output iterátorokat[5]. Legyen például a következő a felépítés:

```
template<typename T> struct
input_iterator_base {
  typedef vect<T> value_type;
  typedef input_iterator_base iterator;
  // ...
  virtual value_type operator*();
  virtual iterator& operator++();
};
// output_iterator_base hasonlóan

template<typename T> struct poly {
  struct input_iterator  : input_iterator_base<T> ;
  struct output_iterator : output_iterator_base<T> ;
  // ...
};
// reg_poly és circle hasonlóan
```

Így bevezethetnénk az következő változókat:

```
class CGTLDrawDoc {
  list<input_iterator_base<scalar>*>  input_iterators;
  list<output_iterátor_base<scalar>*> output_iterators;
  // ...
}
```

De így már felesleges az algoritmusainkat függvény-sablonként implementálni, ugyanis azt, hogy melyik paraméter `input_iterator_base*` és melyik

[5]Valójában célszerűbb lenne iterátor párokat tárolni, de ettől most eltekintünk.

`output_iterator_base*` típusú azt meg tudjuk már a program írásakor mondani. Így például a kirajzoló függvény lehetne a következő[6]:

```
void Draw( list<input_iterator_base<scalar> >& l,
  Graphics& g,
  Pen& p )
{
  list<input_iterator_base<scalar> >::iterator list_iter;
  input_iterator_base iterator save, iter;
  Point p1, p2;

  for(list_iter=l.begin(); list_iter!=l.end; list_iter++)
  {
    save = iter = *list_iter;
    p1 = p2 = ToPoint( *iter );

    do {
      iter++;
      p2 = ToPoint( *iter );
      g.DrawLine(&p, p1, p2);
      p1 = p2;
    } while(iter!=save);
   }
 }
```

Ez már egy kompromisszumos megoldás, de túlzottan, sőt szinte kizárólag az öröklődésre és az objektum orientált programozásra támaszkodik. Egyértelmű ennek a megközelítésnek a hátránya, mivel a iterátor léptetésnél és a `*` operátornál virtuális függvény hívás történik és ezek az alapvető műveletek minden ciklusmagban meghívódnak.

Sablonokkal viszont nem lehet megoldani, kivéve ha nem szimuláljuk valahogy a polimorfizmus, például `void*`-ra konvertálva és megfelelő metaprogramozással biztosítva, hogy mindig a megfelelő függvény hívódjon meg. Egy ilyen jellegű megoldás lehetne a következő:

```
template<typename T> stuct node {
  poly<T>::input_iterator* poly_input_iter;
  poly<T>::output_iterator* poly_output_iter;
  // többi alakzatra hasonlóan
  node* next;
```

[6]Természetesen lehetne általánosabban, függvény-sablonként megvalósítani, de most erre nincs szükség.

```
}
```

De ez nagyon csúnya megoldás lenne, mert igaz, hogy a tárolást megoldaná, és csak ezt a struktúrát kellene módosítani új típusú alakzat hozzáadásakor, de a `node` kiolvasása okozna gondot, azaz egy bonyolult `if/else` elágazáshoz vezetne, ami a polimorf szerű viselkedést biztosítaná. Ez talán megkerülhető lenne meta-programozással, a *Boost.Function*hoz hasonló megoldással.

4.4.5. Konkluzió

Végül gondoljuk át, általánosan, hogy mit is követelünk meg a GTL grafikus könyvtártól. Első sorban azt várjuk el, hogy az alakzatokat hatékonyan ábrázolja és hogy ezekhez az alakzatokhoz biztosítson megfelelő transzformációkat és algoritmusokat, másrészt legyen könnyen bővíthető az alakzatok készlete és az algoritmusok készlete is. Ezeket lényegében teljesíti is a GTL.

Ami hiányzik belőle, az a lehetőség, hogy az alakzatokat közös konténerbe lehessen helyezni és ezt fordítás-idejű, statikus polimorfizmussal, amit a generikus programozás nyújt, nem lehet megoldani. Tegyük fel, ugyanis, hogy van egy olyan „ügyes konténerünk", ami el tudná tárolni az alakzatokat és statikus polimorfizmusra támaszkodik. Ez azt jelentené, hogy fordítás-időben el tudja dönteni, hogy milyen alakzattal fog bővülni. Ez viszont egy rajzoló programban megengedhetetlen, ugyanis a rajzoló programban, a felhasználó csak futás-időben fogja megadni, hogy milyen alakzatokat fog rajzolni. Elég csak az első elemét megfigyelni ennek az „ügyes konténernek", még ha valahogy, el is tudja tárolni az alakzatot, és vissza is tudja adni, akkor is egy `if` elágazással le kell kezelni külön-külön mindegyik alakzat esetét, mivel az volt a feltevés, hogy az alakzatok csak statikusan polimorfak és nincs közös bázisosztályuk.

Ha belegondolunk, világos, hogy dinamikus polimorfizmusra van szükségünk. Az öröklődés és a virtuális függvények pontosan azt a problémát oldják meg amivel szembesülünk. A virtuális függvények, abból a célból tervezték, hogy ha egy objektum típusának és viselkedésének mivoltát, el akarjuk halasztani futási-időre, akkor ne kelljen bonyolult elágazás struktúrákat írni.

Ennek a döntésnek az elhalasztásának ára van. Programozásban, mindig amikor későbbre halasztunk egy döntés, azt valamivel kompenzálni kell. Ha csak futás időben dől el, hogy melyik függvény fog meghívódni, akkor azt vagy egy függvény mutatóban (virtuális függvény táblázat) vagy egy típus azonosítóban kell tárolni és utána konvertálni. A naiv programozó utópiája, hogy egy olyan programot készít amelyen csak egy nyomógomb van, és ha rákattint az ember, azt csinálja a program amit az ember kíván. Ez egy illúzió. Ha nem mindig ugyanazt a rajzot szeretném rajzolni, vagy nem

egy véletlenszerűen generált rajzok akarok, hanem tényleg egy olyan rajzot amit én rajzoltam, akkor azt valahogy a program használatakor, futás-időben kell a programnak kirajzolnia, nem tolható el a döntés fordítás-időbe. Lehet, hogy csak köröket fogunk rajzolni, de a fordító ezt nem tudhatja és a sokszögeket kezelő kódot is le kell, hogy fordítja és azokat az esetek is le kell, hogy legyenek kezelve benne, hogy mi van a ha sokszögre kattintunk és azt akarjuk elforgatni. Ezt meta-programozással nem tudjuk elérni, mivel a meta-programozás lényegében a fordító programból egy interpretert csinál (meta-program egy olyan program amely a fordítás-időben fut le) és ennek a meta-programnak az eredményeit futtatás előtt rögzíti.

5. Alternatív megoldások

Azt beláttuk, hogy kizárólag sablonokat használva nem tudjuk megvalósítani a dinamikus polimorfizmust és ezért nem is tudunk egy hatékony rajzoló programot ha csak ezekre az eszközökre támaszkodunk. Viszont ha gyengítenénk egy kicsit a feltételeinken és megengednénk az öröklődést és a virtuális függvényeket vagy egy ezzel ekvivalens mechanizmust, akkor egy kellően hatékony és elegáns megoldás kapnánk.

A felmerült probléma nem egyedi és sok hasonló problémát már megpróbáltak megoldani valamilyen módon. Ebből adódóan sok más generikus könyvtár született és ezeket fogjuk megvizsgálni, mennyire lennének alkalmasak, hogy megoldják a GTL-el kapcsolatban felmerült problémákat.

5.1. CGAL

Computational Geometry Algorithms Library – röviden CGAL – egy C++-ban írt geometriai algoritmusok könyvtára. Három fő részre lehet bontani:

- Az első rész tartalmazza az úgynevezett *Kernel*eket és aműveleteket amelyeket végre lehet rajtuk hajtani. A kernelek állandóméretű, módosíthatatlan alap objektumok, mint például pontok, egyenesek, síkok stb. A kernelek paraméterezése határozza meg a számtestet amit a pontok koordinátáinak reprezentálására fogunk használni, továbbá megadható hogy 2, 3 vagy n dimenzióban szeretnénk dolgozni és hogy Euklideszi vagy baricentrikus koordinátákkal dolgozunk.

- A második rész alapvető geometriai adatstruktúrákból és algoritmusokból áll. Ezeket az adatstruktúrákat, vagyis alakzatokat a megfelelő kernelekkel, mint trait-ekkel lehet paraméterzeni, ezzel meghatározva azt, hogy milyen alap objektumokkal reprezentáljuk az alakzatokat és

az interfészt az alakzatok és az algoritmusok között. A CGAL nagyon sok geometriai és grafikában használt adatstruktúrát és algoritmust tartalmaz.

- A CGAL harmadik része biztosítja a hatékony implementációt és más nem-geometriai funkcionalitást. Ebben a részben találhatók a CGAL STL kiterjesztések, nyelek (handle), ciklikus iterátorok (circulator), objektum generátorok, időzítők, I/O folyam operátorok, numerikus típusok támogatása, stb.

A CGAL már egy jól kifejlesztett geometriai könyvtár (jelenleg a 3.2.1 verziószámnál tart), viszont nem teljesen oldja meg a polimorfizmus problémáját, ugyanis egy `CGAL::Object` típusú objektum biztosítja a polimorf tárolását az alakzatoknak. Két alap objektum metszete meghatározása nem egyértelmű, hogy mit ad vissza ezért az `intersection` függvény polimorf visszatérési értékű kell, hogy legyen.

```
{
  typedef Cartesian<double>  K;
  typedef K::Point_2         Point_2;
  typedef K::Segment_2       Segment_2;

  Segment_2 segment_1, segment_2;

  std::cin >> segment_1 >> segment_2;

  Object obj = intersection(segment_1, segment_2);

  if (const Point_2 *point = object_cast<Point_2>(&obj)) {
      /* do something with *point */
  } else if (const Segment_2 *segment =
              object_cast<Segment_2>(&obj)) {
      /* do something with *segment*/
  }

  /*  there was no intersection */
}
```

Igaz, hogy nincs virtuális függvény hívás és a `CGAL::Object` explicit nem bázisosztálya a `Segment_2` és a `Point_2` osztálynak, de viszont nem is igazán polimorfak, ugyanis akkor nem kellene `if` elágazásba tenni őket.

5.2. VTL

Már volt róla szó, hogy valójában nem is az alakzatok polimorfizmusára lenne szükségünk, hanem az iterátorok polimorfizmusa elég lenne. A VTL pontosan ezt valósítja meg, de sajnos nem oly módon ahogy nekünk megfelelne.

A View Template Library (VTL, [11])egy flexibilis a C++ STL-jére helyezhető adapter réteg, amely Gary Powell és Martin Weiser fejlesztése alatt áll. A VTL úgynevezett nézeteket (view) biztosít STL konténerek számára. A nézetet, egy olyan konténer adapterként lehet definiálni, amely egy konténer interfészt biztosít az alatta lévő konténerek

- adatainak egy részéhez,
- újrarendezett adataihoz,
- transzformált adataihoz, vagy
- adatainak megfelelő kombinációjához.

Más szóval, a nézetek alternatív elérési módszert biztosítanak egy vagy több konténer adataihoz. Mivel a nézetek is konténerként viselkednek, ezért nem nehéz őket egymásra építeni. Sablon meta-programozással, azt is megoldották a fejlesztők, hogy a nézetek hatékonyan alkalmazkodjanak az alattuk lévő konténerekhez. Továbbá a smart iterátorokkal összehasonlítva, a nézetek csak smart iterátor gyárak (factory).

A VTL sokféle nézetett biztosít, többek között, ami számunkra érdekes lenne az a konkatenáció (Concatenation View) és az unió nézet (Union View).

A konkatenáció nézet egy konténereket tartalmazó konténert tud kisímítani. Például egy

```
vector<vector< int > > matrix;
```

`int`eket tartalmazó vektorok vektorán, vagyis egy két dimenziós vektoron, úgy megy végig mintha egy dimenziós lenne. Az unió nézet is hasonlóan működik. Az unió két különálló konténer összefűzését teszi lehetővé:

```
{
  vector<int> x(4), y(9);
  // x és y vektor feltöltése

  typedef union_view<vector<int>,vector<int> > union1_type;

  cout << "Union of the original containers:" << endl;
  union1_type u1(x,y);
```

```
  dump(cout, u1); // az unió, vagyis x és y kiírása
}
```

Az utóbbi két nézet azért nem használható a GTL iterátorainak az összefűzésére, mert ezek a nézetek megkövetelik, hogy azonos típusú konténereket főzzünk össze, tehát egy sokszög és egy kör iterátorait nem tudnánk összefűzni.

Két figyelemreméltó nézet, a polimorf (Polymorphic View) és a downcast nézet. A polimorf nézet, a dereferencia operátort alkalmazza, egy mutatókat tartalmazó konténer minden elemére, biztosítva a polimorf viselkedést. A downcast nézet viszont egy megadott típusra próbálja konvertálni a konténer objektumait, ha az öröklődés megengedi. Ezek részben kötődnek a GTL alkalmazásakor felmerült problémákhoz, de nem oldják meg azokat.

5.3. Boost

A *Boost* egy kreatív, inspiráló és rendkívül hatékony C++ könyvtárgyűjtemény. Eddig hivatkoztunk a Boost MPL, StaticAssert és a TypeTraits könyvtárakra, és megemlítettük a Function könyvtárat.

5.3.1. Boost MPL Library

A Boost könyvtárak erősen támaszkodnak generikus és meta-programozásra. A Boost MPL könyvtár, kimondottan olyan elemeket tartalmaz amelyek a meta-programozást segítik. A fizikai mértékegységek típusival számoló példa megvalósítása elképzelhetetlen lenne az MPL segítsége nélkül. Az MPL könyvtárban többek között a következő meta-függvények és meta-függvény osztályok találhatók:

- Egész típusú konstansokat burkoló osztály-sablonok, mint például a `bool_<n>`, `int_<n>`, `long_<n>`, `size_t<n>` és `integral_c<T,n>`, amik a futás-idejű konstansoknak felelnek meg. Ezek minden egész számhoz, egy külön típust rendelnek, ami megkönnyíti a meta-programozást.

  ```
  // int_<5> egy típus nem érték így
  // típus paraméterként is át lehet adni
  static int const five = mpl::int_<5>::value;
  ```

- Típus sorozatok, mint például a STL futás-idejű `vector` megfelelője, amely több típus sorozatából, csinál egy új típust.

  ```
  typedef boost::mpl::vector<
    signed char, short, int, long> signed_types;
  ```

- Egészek sorozatát burkoló osztályok, mint például a `vector_c`, amely több konstans egészek sorozatához, rendel egy típust.

```
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
```

- Fordítás-idejű műveletek típusokkal. A `plus` például, két egészet burkoló osztályból, a két egész összegének a burkoló osztályának típusát adja vissza.

```
BOOST_STATIC_ASSERT((
  mpl::plus< mpl::int_<2>, mpl::int_<3>
  >::type::value == 5
));
```

- A fordítás-idejű sorozatokat transzformáló meta-függvények, mint például a `transform`, ami az STL azonos nevű függvényének, fordítás-idejű megfelelője.

```
struct plus_f
{
  template <class T1, class T2>
  struct apply
  {
    typedef typename mpl::plus<T1,T2>::type type;
  };
};

// ... majd alkalmazva
typedef typename mpl::transform<D1,D2,plus_f>::type dim;
```

Az előző `plus_f` meta-függvény osztály megadható öröklődéssel is:

```
struct plus_f
{
  template <class T1, class T2>
  struct apply
    : mpl::plus<T1,T2> {};
};
```

- Magasabb-rendű függvények, mint például a λ-kalkulusból jól ismert `lambda` meta-függvény és az `apply` meta-függvény. Ezek még egyszerűbbé teszik az előző feladatokat, és különleges *placeholder*ekkel a következő kód írását teszik lehetővé.

```
typedef typename mpl::transform
  <mass,length,mpl::minus<_1,_2> >::type dim;
```

5.3.2. Boost.StaticAssert

A StaticAssert pontosan azt csinálja, amire a neve is utal. A futás-idejű assert, terminálja a programot ha valamilyen feltétel nem teljesül, vagyis ha a program valahogy a programozó által inkonzisztens állapotba lép. A Boost StaticAssert-je, pedig a fordítást terminálja, ha fordításkor valami nem teljesül, vagyis ha a meta-program kerül inkonzisztens állapotba.

5.3.3. Boost.TypeTraits

A Boost TypeTrais könyvtára a traiteket támogatja, ami, az objektum orientált programozásban jól ismert burkolás fogalmának, a meta-programozási megfelelője lehetne. A traitek használatára tipikus példa a `swap` függvény amely, egy egészeket tároló *lista* iterátorán és egy egészeket tároló *vektor* iterátorán, a traitek használata nélkül nem működne.

5.3.4. Boost.Function

A *Function* egy iszonyúan hatékony könyvtár amely varázslatos módon biztosítja a fordítás- és futás-idejű polimorfizmust is. A Function könyvtár függvény objektum burkoló osztály-sablonokból áll. A Boost `function` típusú objektumok a függvény-mutatók általánosítása. A Function könyvtár osztály-sablonjainak meg kell adni egy típus listát, amely egy függvény visszatérési értékének típusát, majd a paramétereinek típusát tartalmazza, vagyis egy függvény szignatúrát ír le. Egy ilyen típusú objektumnak bármilyen függvényszerű objektumot értékül adhatunk, legyen az akár függvény-mutató (függvény-pointer) akár funktor objektum, és a a `boost::function` objektum az elvárt módon fog viselkedni. A function könyvtár a következő kódot teszi lehetővé:

```
// float (int,int) szignatúrájú funktor
struct int_div {
  float operator()(int x, int y) const { return((float)x)/y; };
```

```
};

// float (int,int) szignatúrájú függvény
float mul_ints(int x, int y) { return ((float)x) * y; }

int main()
{

  // boost::function<float (int x, int y)> f;
  // olvashatóbb szintaxis, viszont nem hordozható

  // hordozható változat
  boost::function2<float, int, int> f;

  int choice;
  cin >> choince;
  if( choice == MUL )
    f = & mul_ints; // függvény-mutató
    // a & opcionális
  else
    f = int_div; // funktor

  cout << f( 3, 4 ) << endl;
}
```

A fenti program futás időben dönti el, hogy f szorzás lesz e, vagy osztás, attól függetlenül, hogy az osztás egy osztályként, vagyis `struct` ként van definiálva, amelynek implementálva van a függvényhívó operátora, míg a szorzás egy egyszerű függvényként van definiálva. További érdekesség az a tény, a függvény-mutató és a funktor nem származnak közös ősosztályból, sőt nem is származtatással vannak létrehozva, tehát teljesen független típusú objektumok.

5.4. Multi paradigma programozás és típustörlés

Thomas Becker, a honlapján található cikkében szemlélteti az objektum orientált és a generikus programozás közötti feszültséget. Ez az amivel szembesülünk mi is a GTL alkalmazásánál. A megoldás úgy tűnik, hogy abban rejlik, ha feloldva a feszültséget a generikus és az objektum orientált programozás között, alkalmazzuk mindkét paradigmát egyszerre, és így kapjuk a multi paradigma programozást.

Egy hatékony módszer a típustörlés (type erasure). Erre egy jól ismert példák a Boost.Any és a Boost.Funciont. Becker egy `any_iterator` sablonosztály könyvtárt fejlesztett ki amely megoldhatná a GTL alakzatainak a tárolását, mivel ahogy már korábban is megállapítottuk, elég lenne ha csak az iterátorokat tudnánk tárolni. Tudjuk, hogy minden alakzat iterátora vagy input vagy output iterátor, azaz a következők egyike

```
typename shape<scalar>::input_iterator;
typename shape<scalar>::output_iterátor;
```

ahol `shape` helyet a három alakzat típusának egyike áll. Ez a C++ fordító szempontjából három-három teljesen különböző és egymástól teljesen független típus. Viszont ez a három input és három output iterátor csak az alakzat típusában különbözik egymástól, ez viszont minket egyáltalán nem kellene, hogy érintsen, ugyanis nekünk csak az a fontos, hogy iterálni, dereferenciálni és attól függően, hogy input vagy output iterátorról van-e szó, írni illetve olvasni lehessen a dereferenciált iterátort. Ezt a problémát oldja meg az `any_iterator` a Boost Functionhoz hasonló módon.

```
typedef any_iterator<
  int, // value type
  boost::bidirectional_traversal_tag, // traversal tag.
  int const &, // reference type
  ptrdiff_t // difference type
>
bidirectional_const_iterator_to_int;

bidirectional_const_iterator_to_int
  a_bidirectional_const_iterator_to_int;

std::set<int>::const_iterator i1 = a_set.begin(); // ok
a_bidirectional_const_iterator_to_int = i1;

std::set<int>::iterator i2 = a_set.begin(); // ok
a_bidirectional_const_iterator_to_int = i2;

std::vector<int>::const_iterator i3 = a_vector.begin(); // ok
a_bidirectional_const_iterator_to_int = i3;

std::vector<int>::iterator i4 = a_vector.begin(); // ok
a_bidirectional_const_iterator_to_int = i4;
```

```
// std::vector<std::pair<int, int> >::iterator i5;
// nem ok, value type nem konvertál int-re.
```

A fenti értékadások mint futás-időben történhet, vagyis program inputjától függhetnek.

A GTL alakzatainak iterátorait, az `any_iterátor`ral kompatibilissá téve, az alakzatok iterátorait egy `any_iterator` listában tárolhatnánk, és mivel a Boost Functionhoz hasonlóan, az `any_iterator` is futás-időben polimorf, ezért megfelelő módon fog viselkedni. Az `any_iterator` fő előnye, hogy az iterátorok teljesen függetlenek lehetnek és nincs szükség közös bázis osztályra. Továbbá ezzel a módszerrel is csak $O(m+n)$ műveletet és osztályt kell implementálnunk és az algoritmusokat és az adatszerkezeteket egymástól teljesen függetlenül fejleszthetjük.

6. Záradék

Az objektum orientált és a generikus paradigma elemzése után, mélyebben betekintettünk a sablon meta-programozás lehetőségeibe. Ezek után a GTL könyvtár elemzésével folytattuk. Megpróbáltuk a GTL-t alkalmazva, egy működő rajzprogramot elkészíteni. Ez többé kevésbé sikerült, viszont a megoldás messzemenően nem felelt meg elvárásainknak. Szembesültünk a generikus programozás korlátaival és megpróbáltuk ezeket megkerülni. Arra jutotunk, hogy megkerülhetetlen a dinamikus, futás-idejű polimorfizmus és a statikus, parametrikus polimorfizmus nem elég. Megvizsgáltunk más hasonló problémákat és generikus könyvtárakat amelyek hasonló problémákat oldanak meg és ezeket a könyvtárakat elemeztük. Végül javasoltunk egy alternatív megoldást, amely meta-programozással biztosítja a dinamikus polimorfizmust. Tehát a feltételek enyhítésével, megengedve a dinamikus polimorfizmust, a sablon meta-programozás eszközeivel kifejleszthetők a Boost.Functionhoz hasonló osztály-sablonok amelyek, egyszerre biztosítják a dinamikus polimorfizmus és a generikus programozás által nyújtott típus függetlenséget. Azaz feleslegessé teszik a közös bázisosztályokat és lehetővé teszik, hogy az objektum orientált programozással ellentétben, egy n adatszerkezetet és m algoritmust biztosító könyvtárban csak $O(n+m)$ algoritmust keljen megvalósítani $O(m \cdot n)$ helyett.

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`
Dátum: 2007. május 30.

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

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

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

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

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

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

[12] Vatai Emil: *GTL Draw rajzolóprogram*,
ELTE-IK, Programtervező-matematikus szakon, 2007. májusában bemutat nagyprogram és fejlesztői dokumentációja,
`http://gtldraw.googlecode.com/svn/trunk/`,
Dátum: 2007. április 1.

[13] Thomas Becker: *Type Erasure for C++ Iterators*,
`http://thbecker.net/free_software_utilities`
`/type_erasure_for_cpp_iterators/start_page.html`
Dátum: 2007. június 8.