



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. május 31.

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 Kistelegi 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 kellett használni, az öröklődést és a virtuális függvények használatát teljesen mellőzve. A program elkészítése folyamán, komoly problémákba ütköztünk, melyeket a GTL, vagyis általánosan, a 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 kerülhetők 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.

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 indulnánk ki és ebből származtatnánk a `Polygon`, `Circle` és hasonló osztályokat. A `Shape` osztályban deklaráljuk a virtuális `draw`, `move`, `rotate`, stb. metódusokat, majd a származtatott osztályokban definiáljuk őket és egy `Shape*` típusú, azaz alakzatokra mutató mutatókat tartalmazó tömbbe pakoljuk a rajzlap tartalmát. 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 objektum típusú [4]. Így kézenfekvő volt az a megoldás, hogy az `ArrayList`, a `List`, a `Set` és a `Map` konténerek `Object` típusú elemeket (pontosabban referenciákat) tárolnak. Ennek egyértelmű hátránya volt, hogy a tárolók úgymond „nem tudtak” a tárolt elemek típusáról semmit.

Továbbá amikor használták a konténer elemeit, akkor azok mivel `Object` típusúak voltak, ezért futásiidejű, dinamikus típuskonverziót kellett rajtuk végrehajtani ami valamilyen mértékben rontotta a program teljesítményét. Ezt ki lehetett kerülni új, „típus-tudatos” osztályok létrehozásával. Ilyen lehetne például a `ShapeArray` osztály, amelyhez megírhatjuk a megfelelő `DrawAll(ShapeArray sa)` és `RotateAll(ShapeArray sa, float r)` függvényeket.

2. Generikus programozás és az STL

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), a konténerek mint például a `vector`, a `list`, a `set` stb. mind osztály-sablonok, melyeket a tárolandó elemek típusával paraméterezünk. Ezek mellé, az STL még a megfelelő algoritmusokat is biztosítja, mint például a `sort` vagy a `find`. Az algoritmusok függvény sablonként vannak implementálva, és iterátorokon keresztül „kommunikálnak” a konténerekkel. Az STL-nek megközelítése számos előnye van az `Object`eket tároló konténerrel szemben:

- Típus biztonságos: A fordító mindent elvégez helyettünk. Fordítási időben példányosítja a osztály-sablonokat és függvény-sablonokat, és a megfelelő típus ellenőrzéseket is elvégzi. Tehát kisebb a hiba lehetőség.
- Hatékony: A sablonok segítségével a fordító mindent a fordítási időben kiszámol, leellenőriz és behelyettesít – minden a fordításkor eldől semmit sem hagy futási-időre.
- Egyszerűbb bővíteni és programozni: akár új algoritmusokkal, akár új konténerekkel egyszerűen bővíthetjük a könyvtárat, csak a megfelelő iterátorokat kell implementálni és a megfelelő konvenciókat (concept-eket) követni.

Az utobbiból 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.

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 külön-külön a `IntVector`-t és a `FloatVectort`, a sablonok ezt automatikusan megteszik helyettünk, ugyanis lényegében ugyan azt a kódot kell megírni, csak más típussal.

3. 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 sablon-mechanizmus Turing-teljességét, mint (fordítás idejű) nyelv, megfogant a C++ sablon-metaprogramozás (template metaprogramming) fogalma.

A sablonok, egy a rekurzív függvény híváshoz hasonló elveken működik. A részben specializált sablonok, függvény híváshoz hasonlítanak és a teljes specializációk pedig terminálják a rekurziót. Ezt nagyon jól szemlélteti a következő példa:

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

template <int 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” meghívódik, 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 >> fact;
    std::cout << fact << endl;
    return 0;
}
```

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

Természetesen jogos az észrevétel, hogy minek az ossz vesződés, hogy írjunk egy eleve nehezen érthető és talán félrevezető sablon-metaprogramot ha egy egyszerű kalkulátor programmal, vagy akár papíron kiszámolhatjuk, hogy $15! = 1307674368000$ és az előző példa programunkat a következő, elvileg azonos programra:

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

Ennek több oka is lehet. Első sorban, ha többször akarnánk olyan konstansokat használni a programunkban, ami egy pozitív egész faktoriálisa, akkor egyértelmű, hogy a többszörös számolás és többszörös hiba elkerülése véget, azaz pontossági, helyességi és kényelmi szempontok miatt is megéri egy ilyen kis metaprogram megírása. Ez lehet, hogy a faktoriálisra nem túl sűrűn lesz szükségünk, de például ahogy a C++ nem csak éáőúúőü

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 - Concepts, 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 Meta Template Library