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 lisát választunk, nem kerülhetjük ki a lista típusossá tételéhez szükséges extra programozás, ami egyúttal hibaforrás is. Legalább is ez volt a helyzet a sablonok bevezetése előtt.

1.2. Az első generikus 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), 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. Ez lett az első, és sokáig az egyetlen, hatékony generikus könyvtár implementáció.

Az STL-nek megközelítése számos előnye van az `Object`eket tároló konténerrel szemben:

- Típus biztonságos: A fordító mindent elvégez helyettünk. Fordításiidőben 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 (concepteket) 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 különkülön a `IntVector`-t és a `FloatVectort`, a sablonok ezt automatikusan megteszik helyettünk, ugyanis lényegében ugyan azt a kódot kell megírni, csak más típussal.

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 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 észre venni, 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;
}
```

Hogy ezt a kérdést megvizsgáljunk vegyünk talán egy másik, hasonló példát:

```
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 a következő módon alkalmazhatjuk:

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

Ezzel a kis trükkel, megtehetjük azt, hogy egy számot a bináris alakjában is megadhatunk ha úgy kényelmesebb.

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 Meta Template Library