Programação Genérica

Levando a Abstração ao Limite

Fabio Galuppo, M.Sc.

http://fabiogaluppo.com e http://simplycpp.com/

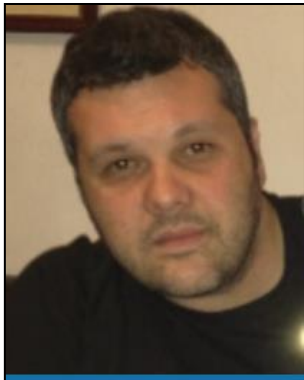
fabiogaluppo@acm.org

@FabioGaluppo

Microsoft MVP Visual Studio and Development Technologies

https://mvp.microsoft.com/en-us/PublicProfile/9529

http://bit.ly/prog_gen_qconsp2016



Award Categories
Visual Studio and Development Technologies

First year awarded:
2002

Number of MVP Awards:
13

Fabio Razzo Galuppo, M.Sc.

Novembro 1973

- Mestrado em Engenharia Elétrica (Universidade Presbiteriana Mackenzie)
  - Ciência da Computação - Inteligência Artificial
- Por mais de 10 anos premiado com Microsoft MVP em Visual C++
- Engenheiro de Software (Programador)
- Matemática Aplicada
- Linguagens de programação prediletas:
  - C++
  - F#
  - Haskell
- Rock'n'Roll
  - E boa música em geral
- http://fabiogaluppo.com
- https://twitter.com/FabioGaluppo
- https://github.com/fabiogaluppo
- http://simplycpp.com

Templates e *Generics*

Polimorfismo Paramétrico

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

  When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

https://docs.oracle.com/javase/tutorial/java/generics/why.html

Templates e *Generics*

Polimorfismo Paramétrico e Restrições

```
public class GenericList<T> where T : Employee
{
    private class Node
    {
        private Node next;
        private T data;

        public Node(T t)
        {
            next = null;
            data = t;
        }
```

Constraint
where T: struct
where T : class
where T : new()
where T : <base class name>
where T : <interface name>
where T : U

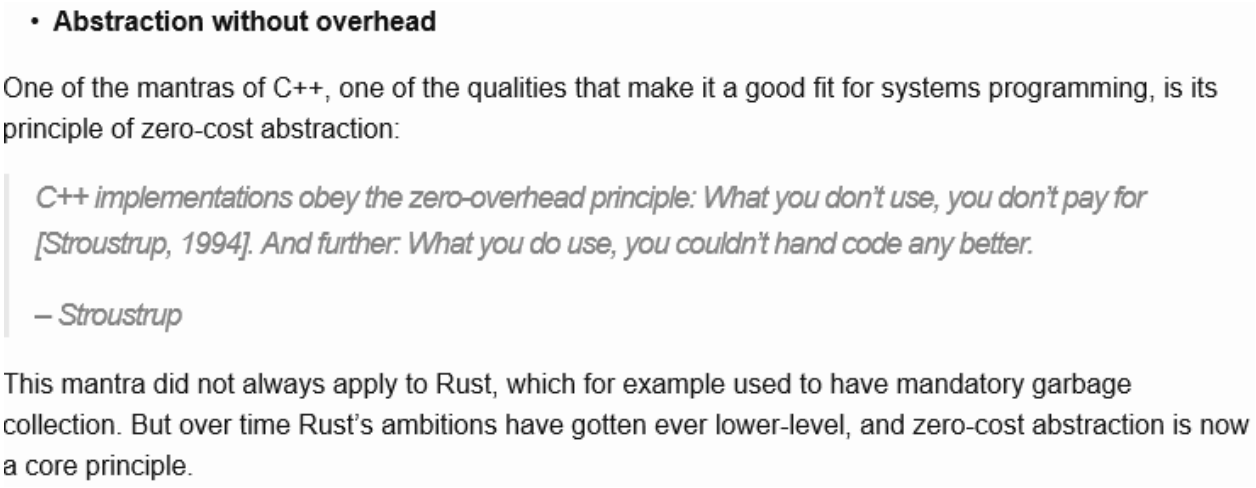
```
class Dictionary<TKey,TValue> :
    IDictionary<TKey,TValue>
    where TKey: IComparable<TKey>
    where TValue: IKeyProvider<TKey>,
                  IPersistable, new()
{
    public void Add(TKey key, TValue value)
    {
        ...
    }
}
```

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

https://msdn.microsoft.com/en-us/library/d5x73970.aspx

Abstrações Custo Zero

- *"The aim of C++ is to help in classical systems programming tasks. It supports the use of light-weight abstraction for resource-constrained and often mission-critical infrastructure applications. The aim is to allow a programmer to work at the highest feasible level of abstraction by providing*
  - *A simple and direct mapping to hardware*
  - *Zero-overhead abstraction mechanisms*
- *In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for"* – Bjarne Stroustrup (http://www.stroustrup.com/ETAPS-corrected-draft.pdf)



- **Abstraction without overhead**

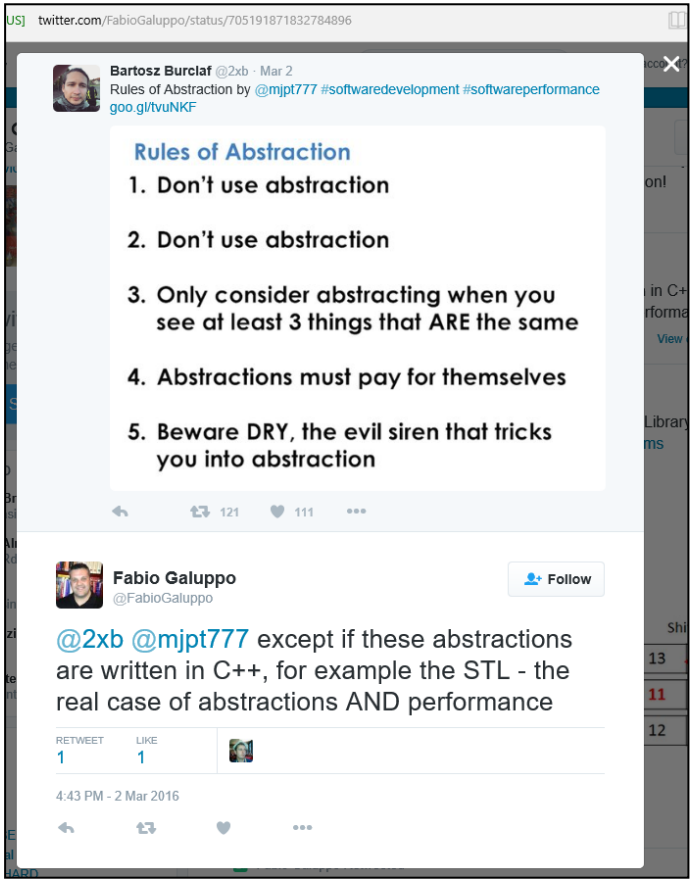
One of the mantras of C++, one of the qualities that make it a good fit for systems programming, is its principle of zero-cost abstraction:

> *C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [Stroustrup, 1994]. And further: What you do use, you couldn't hand code any better.*
>
> *– Stroustrup*

This mantra did not always apply to Rust, which for example used to have mandatory garbage collection. But over time Rust's ambitions have gotten ever lower-level, and zero-cost abstraction is now a core principle.

http://blog.rust-lang.org/2015/05/11/traits.html

Quando o contexto é sobre desempenho



https://twitter.com/FabioGaluppo/status/705191871832784896

- Não use abstração (! ou ?)

O que é Programação Genérica?

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Generic Programming (David Musser, Alexander Stepanov)

Generic programming is about making programming languages more flexible without compromising safety. Both sides of this equation are important, and becoming more so as we seek to do more and more with computer systems, while becoming ever more dependent on their reliability.

The term 'generic programming' means different things to different people, because they have different ideas about how to achieve the common goal of combining flexibility and safety. To some people, it means *parametric polymorphism*; to others, it means libraries of *algorithms and data structures*; to another group, it means *reflection and meta-programming*; to us, it means *polytypism*, that is, type-safe parametrization by a datatype. Rather than trying to impose our mean-

Datatype-Generic Programming (Jeremy Gibbons)

Abstraindo do Concreto

$$
\begin{aligned}
&stringToUpper :: List\ Char \rightarrow List\ Char \\
&stringToUpper\ Nil \qquad\qquad = Nil \\
&stringToUpper\ (Cons\ x\ xs) = Cons\ (toUpper\ x)\ (stringToUpper\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&classifyAges :: List\ Integer \rightarrow List\ Bool \\
&classifyAges\ Nil \qquad\qquad = Nil \\
&classifyAges\ (Cons\ x\ xs) = Cons\ (x < 30)\ (classifyAges\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&mapL :: (a \rightarrow b) \rightarrow (List\ a \rightarrow List\ b) \\
&mapL\ f\ Nil \qquad\qquad = Nil \\
&mapL\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (mapL\ f\ xs)
\end{aligned}
$$

Datatype-Generic Programming (Jeremy Gibbons)

Abstraindo do Concreto

$$
\begin{aligned}
&append :: List\ a \rightarrow List\ a \rightarrow List\ a \\
&append\ Nil\ \ \ \ \ \ \ \ \ \ ys = ys \\
&append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)
\end{aligned}
$$

$$
\begin{aligned}
&concat :: List\ (List\ a) \rightarrow List\ a \\
&concat\ Nil\ \ \ \ \ \ \ \ \ \ \ \ = Nil \\
&concat\ (Cons\ xs\ xss) = append\ xs\ (concat\ xss)
\end{aligned}
$$

$$
\begin{aligned}
&sum :: List\ Integer \rightarrow Integer \\
&sum\ Nil\ \ \ \ \ \ \ \ \ \ \ \ = 0 \\
&sum\ (Cons\ x\ xs) = x + sum\ xs
\end{aligned}
$$

$$
\begin{aligned}
&foldL :: b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow List\ a \rightarrow b \\
&foldL\ n\ c\ Nil\ \ \ \ \ \ \ \ \ \ \ \ = n \\
&foldL\ n\ c\ (Cons\ x\ xs) = c\ x\ (foldL\ n\ c\ xs)
\end{aligned}
$$

Datatype-Generic Programming (Jeremy Gibbons)

Então, o que é mesmo Programação Genérica?

Generic Programming is an

- approach to programming…
- focused on designing algorithms and data structures so that they work in the most general setting…
- without loss of efficiency

- Generic programming is more of an *attitude* than a particular set of tools

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt1.pdf

Álgebra Abstrata e Teoria dos Números

Generic Programming Approach

Abstract Algebra: Branch of mathematics concerned with reasoning about entities in terms of abstract properties of operations on them

- When you're trying to find the most general way to express an algorithm, or a mathematical idea, you need to start with a concrete problem and concrete examples
- In mathematics, the concrete problems that drove abstract algebra come from *number theory*.

Number Theory: Branch of mathematics that deals with properties of integers, especially divisibility.

$$x^n + y^n = z^n \quad [n > 2]$$

https://www.youtube.com/watch?v=ReOQ300AcSU

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt1.pdf

Álgebra Abstrata e Estruturas Algébricas

Branch of mathematics that deals with abstract entities called *algebraic structures*

– Collections of objects that follow certain rules

Abstract algebra lets us prove results for structures such as groups *without knowing anything about either the items in the group or the operation*.

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt2.pdf

Estrutura Algébrica: Grupo

A *group* is a set on which the following are defined:

- Operations: $x \circ y, x^{-1}$
- Constant: e

and for which the following axioms hold:

- Associativity $x \circ (y \circ z) = (x \circ y) \circ z$
- Identity $x \circ e = e \circ x = x$
- Cancellation $x \circ x^{-1} = x^{-1} \circ x = e$

- The group operation is not necessarily commutative, i.e. it is not necessarily true that $x \circ y = y \circ x$.

- Grupo implica:
  - Operação Binária Associativa
  - Inversa e Cancelamento
  - Elemento de Identidade

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt2.pdf

Associatividade

s.f. Matemática Propriedade de uma lei de composição interna na qual se pode substituir a sucessão de certos elementos pelo resultado da operação efetuada com eles sem alterar o resultado global.

http://www.dicio.com.br/associatividade/

$$a \odot b \odot c \odot d \odot e \odot f \odot g \odot h$$

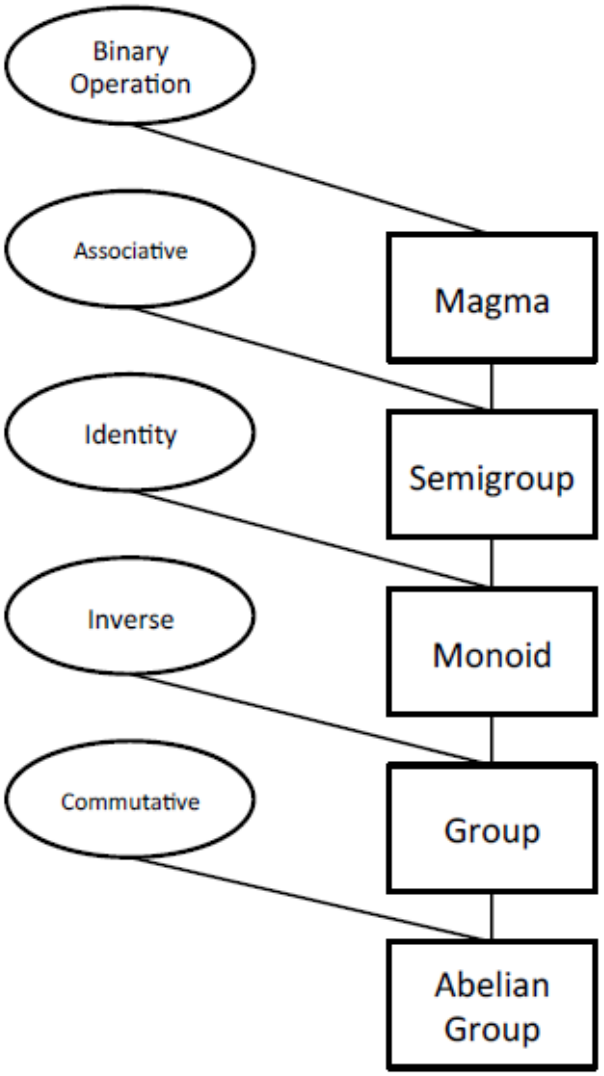
$$= ((((((a \odot b) \odot c) \odot d) \odot e) \odot f) \odot g) \odot h$$

$$= a \odot (b \odot (c \odot (d \odot (e \odot (f \odot (g \odot h))))))$$

$$= ((a \odot b) \odot (c \odot d)) \odot ((e \odot f) \odot (g \odot g))$$

AlexFest: Paul McJones - The Concept of Concept

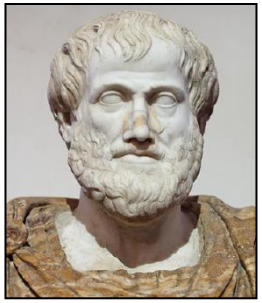
Estruturas Algébricas



http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt2.pdf

Abstração, revisitada

- We've been discussing abstraction in mathematics:
  - abstract algebra
- Next, we'll look at abstraction in programming:
  - generic programming and the notion of *concepts*
- Where did these ideas about abstraction originate?
  - Aristotle



http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

Valores e Tipos

- A *datum* is a sequence of bits.
  - Example: 01000001
- A *value* is a datum together with its interpretation.
  - Example: 01000001 interpreted as the character 'A'
  - Example: 01000001 interpreted as the integer 65
  - A datum without an interpretation has no meaning
- A *value type* is a set of values sharing a common interpretation.

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

Objetos

- An *object* is a collection of bits in memory that contain a value of a given value type.
  - There is no requirement that bits of an object be contiguous. Many objects have *remote parts*.
  - An object is *immutable* if its value never changes, and *mutable* otherwise.
  - An object is *unrestricted* if it can contain any value of its value type.
    - An *object type* is a uniform method of storing and retrieving values of a given value type from a particular object, given its address.
      - What we call "types" in programming languages are object types.

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

Concepts

A concept in programming is:

- a way to describe a family of related object types
- (Equivalently) a set of requirements on types

Concepts are to types what types are to instances.

Mathematics	Programming	Programming Examples
Theory	Concept	**Integral, Character**
Model	Type or Class	`uint8_t`, `char`
Element	Instance	01000001 (65, 'A')

- **Integral**
  - `int8_t, uint8_t, int16_t,…`
- **UnsignedIntegral**
  - `uint8_t, uint16_t, …`
- **SignedIntegral**
  - `int8_t, int16_t, …`

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

The Concept of *Concept*

Abstraction	*Data type*	*Concept, abstract algorithm*
What it is	Interface (specification, encapsulated implementation)	Semantic properties, algorithms they enable
Focus	Data structures	Algorithms
What's protected	Representation invariant	Generality of algorithm
Who	Parnas, Hoare, Liskov & Zilles, Guttag, Musser, … (870 papers by 1983)	Stepanov and his collaborators: Kapur, Musser, Kershenbaum, Lee; Scheme, Ada, C++

AlexFest: Paul McJones - The Concept of Concept

O Conceito de Tipo Regular

- Operations Requirements

A type is *regular* if it supports these operations:

- copy construction
- assignment
- equality
- destruction

Having a copy constructor implies having a default constructor, since

```
T a(b);
```

should be equivalent to:

```
T a; a = b;
```

- Semantic Requirements

$$\forall a\ \forall b\ \forall c\ : T\ a(b) \implies (b = c \implies a = c)$$
$$\forall a\ \forall b\ \forall c\ : a \leftarrow b \implies (b = c \implies a = c)$$
$$\forall f \in \text{RegularFunction}\ : a = b \implies f(a) = f(b)$$

- Space/Time Complexity Requirements
  - Each required operation must be no worse than linear in the area of the object

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

Standard Template Library (STL)

The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

http://www.sgi.com/tech/stl/stl_introduction.html

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

http://www.cplusplus.com/reference/stl/

Regular types can be stored in STL containers

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt2.pdf

Iterators

An *iterator* is a concept used to express position in a sequence.

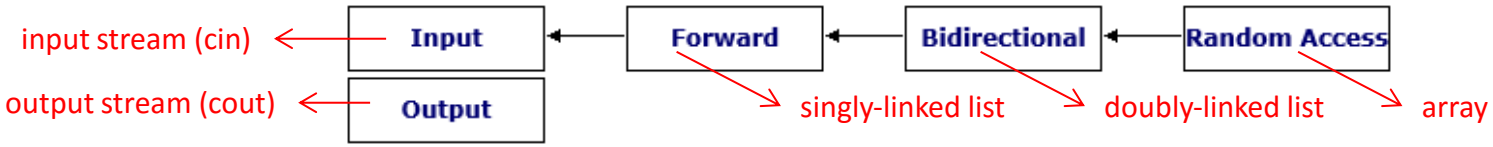
- Iterators are a generalization of pointers.
- A better name for STL iterators would have been “position” or “coordinate.”
- An iterator is “something that lets you do linear search in linear time.”

Required Operations:

- Regular type operations
- Successor
- Dereference

http://www.fm2gp.com/slides/FM2GP_Course_Slides_Pt3.pdf

Categoria de *Iterators*



category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and –	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

http://www.cplusplus.com/reference/iterator/

Algoritmos

s.m. Matemática Sequência de raciocínios ou operações que oferece a solução de certos problemas.

http://www.dicio.com.br/algoritmo/

STL Algorithms

http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4567.pdf

- Compreensível e mais abstraido do que um *raw loop*
- Mantém *side-effects* dentro de uma interface bem definida
- Facilita o raciocínio sobre o problema
- Atua em conjunto com *iterators* ou *left-closed interval* $[begin, end)$

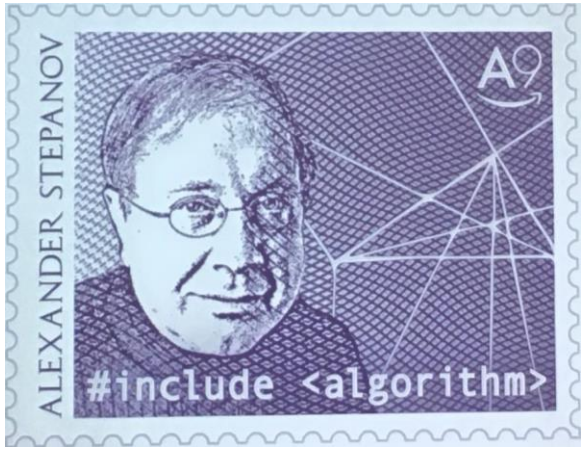
Policy-based Design

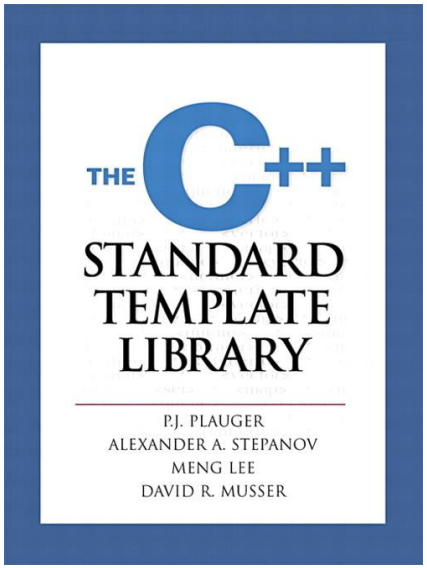
Policy-based design, also known as **policy-based class design** or **policy-based programming**, is a computer programming paradigm based on an idiom for C++ known as **policies**. It has been described as a compile-time variant of the strategy pattern, and has connections with C++ template metaprogramming. It was first popularized by Andrei Alexandrescu with his 2001 book *Modern C++ Design* and his column *Generic<Programming>* in the *C/C++ Users Journal*.

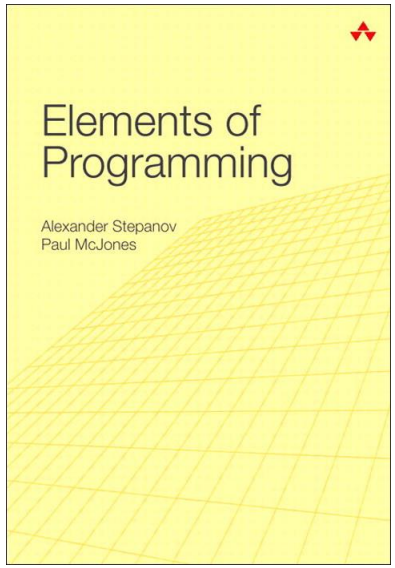
The central idiom in policy-based design is a class template (called the *host* class), taking several type parameters as input, which are instantiated with types selected by the user (called *policy classes*), each implementing a particular implicit interface (called a *policy*), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class. By supplying a host class combined with a set of different, canned implementations for each policy, a library or module can support an exponential number of different behavior combinations, resolved at compile time, and selected by mixing and matching the different supplied policy classes in the instantiation of the host class template. Additionally, by writing a custom implementation of a given policy, a policy-based library can be used in situations requiring behaviors unforeseen by the library implementor. Even in cases where no more than one implementation of each policy will ever be used, decomposing a class into policies can aid the design process, by increasing modularity and highlighting exactly where orthogonal design decisions have been made.

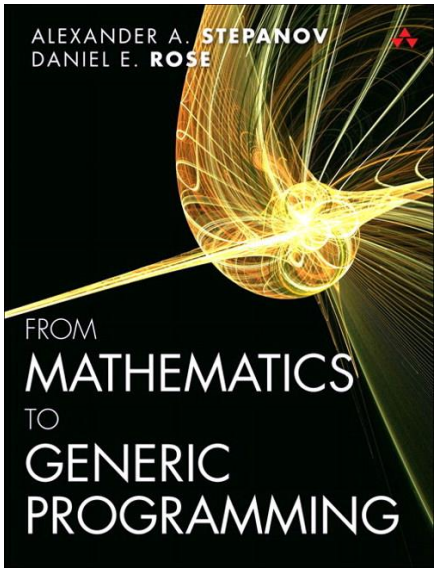
https://en.wikipedia.org/wiki/Policy-based_design

Alexander Stepanov









Alex Stepanov

By Bjarne Stroustrup | Jan 21, 2016 07:49 AM | News | Tags: *None*

Save to: Instapaper Pocket Readability

Alex Stepanov retired last week. He's one of the most prominent members of the C++ community and one of the most innovative contributors to the C++ standard. He was the father of the STL and probably the first promotor of "concepts" as we now know them. Concepts, as specified in the ISO TS, will ship as part of GCC6.0 "any day now." His work on generic programming goes back in time through Ada (1987), Scheme (1986), and Tecton (1981). See his list of contributions (books, articles, talks, and videos): http://www.stepanovpapers.com/. Without him, we would not have had generic programming as we know it and C++ would have been a very different and poorer language.

https://isocpp.org/blog/2016/01/alex-stepanov

Alexander Stepanov, thank you very much!

http://www.stepanovpapers.com/

Final Thought

- “Programming is an iterative process: studying useful problems, finding efficient algorithms for them, distilling the concepts underlying the algorithms, and organizing the concepts and algorithms into a coherent mathematical theory. Each new discovery adds to the permanent body of knowledge, but each has its limitations.”

  AlexFest: Paul McJones - The Concept of Concept

- “STL is intended to be an example. An example of how you code, not the beginning, not an end, it's an example.”

  CppCon 2015: Sean Parent "Better Code: Data Structures"

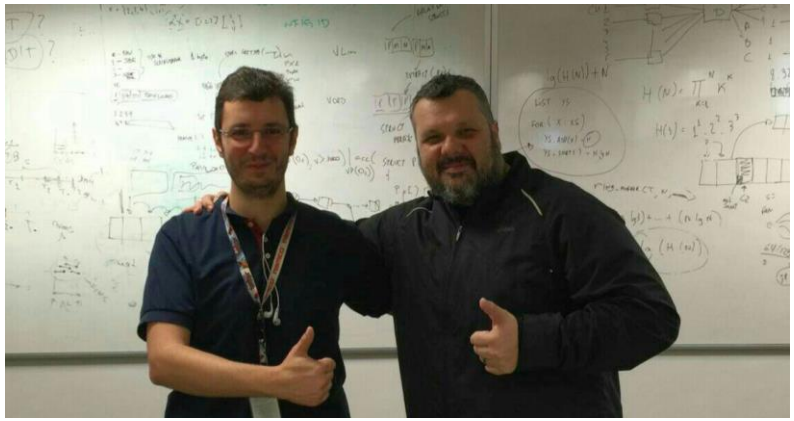
- “Abstração é essencial. Você não precisa abrir mão dela para alto desempenho, basta fazê-la da forma correta, de preferência com a linguagem de programação certa para isso.” – Fabio Galuppo, QCon SP 2016

Se quiser saber mais sobre:

C++
Programação Genérica
Iterators
Policy-based Design
Algoritmos
...

visite:

www.simplycpp.com



http://www.simplycpp.com

Programação Genérica

Levando a Abstração ao Limite

Fabio Galuppo, M.Sc.

http://fabiogaluppo.com e http://simplycpp.com/

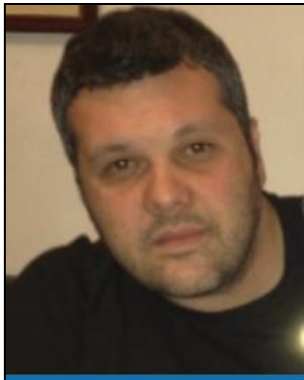
fabiogaluppo@acm.org

@FabioGaluppo

Microsoft MVP Visual Studio and Development Technologies

https://mvp.microsoft.com/en-us/PublicProfile/9529

http://bit.ly/prog_gen_qconsp2016



Award Categories
Visual Studio and Development Technologies

First year awarded:
2002

Number of MVP Awards:
13