C++ templates

From algebra to disassembly

On GitHub: https://github.com/akalenuk/templates-experiments
On Medium (soon): https://medium.com/@okaleniuk

Let's start with the magic of algebra

$$2ax + 3a + bc = cb \Leftrightarrow 2ax + 3a + bc = bc$$

$$2ax + 3a + bc = bc \Leftrightarrow 2ax + 3a = 0$$

$$2ax + 3a = 0 \Leftrightarrow 2ax = -3a$$

$$2ax = -3a \Leftrightarrow 2x = -3$$

$$2x = -3 \Leftrightarrow x = -1.5$$

Seems legit, doesn't it?

But what did I do wrong though?

I didn’t specify what are a, b, c and x

If these are all integers, then obviously this wouldn’t work

$$a/b = c/b \not\Rightarrow a = c$$

$$2/2 = 3/2 \not\Rightarrow 2 = 3$$

Even worse for natural

$$a \in \mathbb{N}, b \in \mathbb{N} \not\Rightarrow a - b \in \mathbb{N}$$

And for quaternion multiplication is not even commutative

$$a \in \mathbb{H}, b \in \mathbb{H} \not\Rightarrow ab = ba$$

Three things to put together

1) Different types of numbers have different algebras
2) Not only *numbers*, but *tensors*, *vectors*, *functions*, *transformations*, *symmetries*, – basically every type of mathematical object you can think out an operation or few for may have its own algebra.
3) Some of these objects share algebraic properties. All or partially.

Why is this awesome?

This is Bob.
Bob spent years proving theorems for square matrices

This is Bob when he learned that square matrices don’t attract funding anymore.

This is Bob when he learned that *square matrices* and *integers modulo X* share the same algebra.
Bob spent a night doing “Ctrl-c + Ctrl-v” and he is now world class specialist in uint32_t.

That's why abstract algebra is awesome!

Knowing that the basic properties of algebraic systems are similar, you may presume that the derived properties are held as well.

You can prove one theorem (or propose an algorithm) for one algebraic system, and it will automagically work for another!

Sounds familiar?

C++ templates

```cpp
#include <iostream>

template <typename T>
T doubled_1(const T& i_one){
        return i_one * 2;
}

template <typename T>
T doubled_2(const T& i_one){
        return i_one + i_one;
}

#ifdef WE_WANT_A_COMPILER_ERROR
template <typename T>
T doubled_3(const T& i_one){
        blah_blah i_one + i_one;
}
#endif

template <typename T>
T doubled_4(const T& i_one){
        return i_one.blah_blah;
}

int main(){
        std::cout << doubled_1(1) << std::endl;
        std::cout << doubled_1(1u) << std::endl;
        std::cout << doubled_1<float>(1.) << std::endl;

        std::cout << doubled_2(std::string("1")) << std::endl;
}
```

Compiler checks

- for syntax errors – always;
- for type consistency – on instantiation;
- for algebraic properties – never.

It turns 1 into 11 for strings. Let’s not do that.

Specialize and delete

```cpp
#include <iostream>


template <typename T>
T doubled(const T& i_one){
        return i_one + i_one;
}

std::string doubled(const std::string& i_one) = delete;

int main(){
        std::cout << doubled(1) << std::endl;
        std::cout << doubled(1u) << std::endl;
        std::cout << doubled<float>(1.) << std::endl;

#ifdef WE_WANT_A_COMPILER_ERROR
        std::cout << doubled(std::string("1")) << std::endl;
#endif
}
```

Technically we can specialize a function per every type but...
what is the point of templates then?

And it gets worse with more parameters.

Partial specialize and delete

```
#include <iostream>


template <typename T1, typename T2>
T1 added(const T1& i_one, const T2& i_two){
        return i_one + i_two;
}

template <typename T1>
T1 added(const T1& i_one, const std::string& i_two) = delete;

template <typename T2>
std::string added(const std::string& i_one, const T2& i_two) = delete;

std::string added(const std::string& i_one, const std::string i_two) = delete;

int main(){
        std::cout << added(1, 2) << std::endl;

#ifdef WE_WANT_A_COMPILER_ERROR
        std::cout << added(std::string("1"), std::string("2")) << std::endl;
        std::cout << added(1, std::string("2")) << std::endl;
        std::cout << added(std::string("1"), 2) << std::endl;
#endif
}
```

Even for a few parameters things grow ugly fast.

Although, there is a way to deal with how many parameters you want.

Partial specialize with variadic templates

```
#include <iostream>

int added(){
        return 0;
}

template <typename Head, typename... Tail>
Head added(const Head& i_head, Tail... i_tail){
        return i_head + added(i_tail...);
}

template <typename... Tail>
std::string added(const std::string& i_head, Tail... i_tail) = delete;

int main(){
        std::cout << added(1.f, 2u, 3l) << std::endl;
#ifdef WE_WANT_A_COMPILER_ERROR
        std::cout << added(std::string("1"), 2u, 3l) << std::endl;
#endif
}
```

They fell out of fashion. Boost for one favors operator overloads and explicit chain of execution.

But we already have variadic macroses and functions,
so why not templates?

Type traits

```
#include <iostream>
#include <type_traits>

template <typename T>
T doubled(const T& i_one, typename std::enable_if<std::is_floating_point<T>::value >::type* = 0){
        return i_one + i_one;
}

template <typename T>
T doubled(const T& i_one, typename std::enable_if<std::is_integral<T>::value >::type* = 0){
        return i_one + i_one;
}

int main(){
        std::cout << doubled(1) << std::endl;
        std::cout << doubled(1u) << std::endl;
        std::cout << doubled<float>(1.) << std::endl;

#ifdef WE_WANT_A_COMPILER_ERROR
        std::cout << doubled(std::string("1")) << std::endl;
#endif
}
```

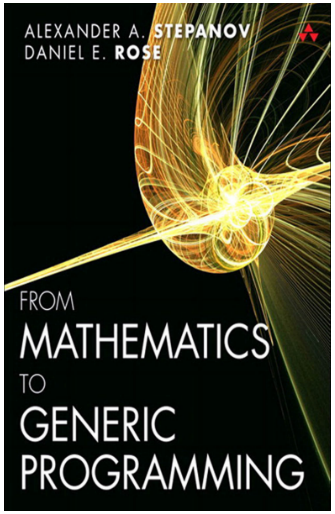
Better, but still it’s either code duplication,
or very elaborate type traits system

Concepts

```
template <InputIterator I, Predicate P>
std::pair<I, DifferenceType<I>>
find_if_n(I f, DifferenceType<I> n, P p) {
    while (n && !p(*f)) { ++f; --n; }
    return {f, n};
}
```

Classes for types (similar to Haskell *typeclass*)
Beautiful in every way, but they are not in the standard.

An excellent read on the topic!



Parametric types

```
template<typename T>
class Vector {
private:
    T* elem;   // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Vector(int s);          // constructor: establish invariant, acquire resources
    ~Vector() { delete[] elem; }     // destructor: release resources

    // ... copy and move operations ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const { return sz; }
};
```

This doesn’t work flawless either. Remember *vector<auto_ptr>*.
Or even *vector<bool>*.

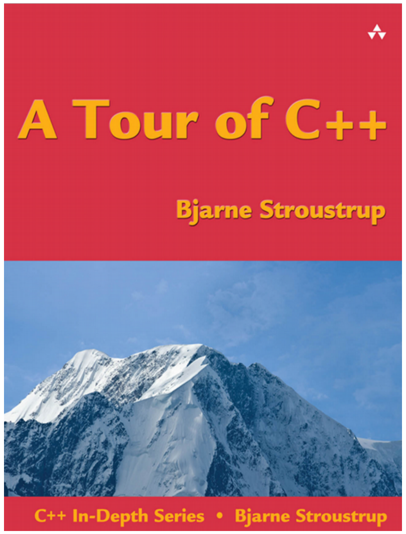
Dependent names and aliases

```
template<typename C>
using Element_type = typename C::value_type;     // the type of C's elements

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec;        // keep results here
        // ...
}
```

Function objects

```
template<typename T>
class Less_than {
        const T val;      // value to compare against
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; }  // call operator
};
```



Non-type parameters

Work basically like
type parameters.

Every instantiation
makes compiler generate
another piece of code.

But! Compilers are surprisingly
good in reducing redundancies.

```cpp
template <class T, unsigned int RADIX_BITS> struct Trie{
    constexpr static unsigned int mask(unsigned int radix_bits){
        return (radix_bits == 1) ? 1 : (1 + (mask(radix_bits - 1) << 1));
    }
    constexpr static unsigned int pow_of_2(unsigned int exp){
        return (exp == 1) ? 2 : (2*pow_of_2(exp - 1));
    }
    constexpr static unsigned int steps_in_byte = 8 / RADIX_BITS;

    std::vector<Trie*> subtries;
    T value;

    Trie(){
        subtries.resize( pow_of_2(RADIX_BITS), nullptr );
    }

    ~Trie(){
        for(auto* trie : subtries)
            delete trie;
    }

    void set(const char* key, T value){
        Trie* trie = this;
        while(key[0] != '\0'){
            char c = key[0];
            for(unsigned int i = 0; i < steps_in_byte; i++){
                int radix0 = c & mask(RADIX_BITS);
                c = c >> RADIX_BITS;
                if(trie->subtries[radix0] == nullptr)
                    trie->subtries[radix0] = new Trie();
                trie = trie->subtries[radix0];
            }
            key++;
        }
        trie->value = value;
    }

    T get(const char* key){
        Trie* trie = this;
        while(key[0] != '\0'){
            char c = key[0];
            for(unsigned int i = 0; i < steps_in_byte; i++){
                int radix0 = c & mask(RADIX_BITS);
                c = c >> RADIX_BITS;
                trie = trie->subtries[radix0];
            }
            key++;
        }
        return trie->value;
    }
};
```

Accidental meta-programming

```cpp
#include <iostream>
#include <array>

template <size_t J, size_t I, size_t N>
static inline void inner_loop(std::array<int, N>& a){
    if(J < N-I-1){
        int d = std::abs(a[J]-a[J+1]);
        int s = a[J] + a[J+1];
        a[J] = (s-d) / 2;
        a[J+1] = (s+d) / 2;
        inner_loop<J + (J<N-I-1), I, N>(a);
    }
}

template <size_t I, size_t N>
static inline void outer_loop(std::array<int, N>& a){
    if(I < N - 1){
        inner_loop<0, I, N>(a);
        outer_loop<I + (I<N-1), N>(a);
    }
}

template <size_t N>
void static_sort(std::array<int, N>& a){
    outer_loop<0, N>(a);
}

int main()
{
    auto a = std::array<int, 8> {6,5,4,7,3,5,1,2};
    static_sort(a);
    for(auto ai : a)
        std::cout << ai << std::endl;
}
```

```asm
	.file		"static-sort.cpp"
	.section	.rodata.str1.1,"aMS",@progbits,1
.LC2:
	.string		" "
	.section	.text.unlikely,"ax",@progbits
.LCOLDB3:
	.section	.text.startup,"ax",@progbits
.LHOTB3:
	.p2align 4,,15
	.globl		main
	.type		main, @function
main:
.LFB1497:
	.cfi_startproc
	pushq		%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	pushq		%rbx
	.cfi_def_cfa_offset 24
	.cfi_offset 3, -24
	subq		$56, %rsp
	.cfi_def_cfa_offset 80
	movdqa		.LC0(%rip), %xmm0
	leaq		32(%rsp), %rbp
	movq		%rsp, %rbx
	movq		%fs:40, %rax
	movq		%rax, 40(%rsp)
	xorl		%eax, %eax
	movaps		%xmm0, (%rsp)
	movdqa		.LC1(%rip), %xmm0
	movaps		%xmm0, 16(%rsp)
	.p2align 4,,10
	.p2align 3
.L2:
	movl		(%rbx), %esi
	movl		$_ZSt4cout, %edi
	addq		$4, %rbx
	call		_ZNSolsEi
	movl		$1, %edx
	movl		$.LC2, %esi
	movq		%rax, %rdi
	call		_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
	cmpq		%rbp, %rbx
	jne		.L2
	xorl		%eax, %eax
	movq		40(%rsp), %rcx
	xorq		%fs:40, %rcx
	jne		.L7
	addq		$56, %rsp
	.cfi_remember_state
	.cfi_def_cfa_offset 24
	popq		%rbx
	.cfi_def_cfa_offset 16
	popq		%rbp
	.cfi_def_cfa_offset 8
	ret
.L7:
	.cfi_restore_state
	call		__stack_chk_fail
	.cfi_endproc
.LFE1497:
	.size		main, .-main
	.section	.text.unlikely
.LCOLDE3:
	.section	.text.startup
.LHOTE3:
	.section	.text.unlikely
.LCOLDB4:
	.section	.text.startup
.LHOTB4:
	.p2align 4,,15
	.type		_GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB1733:
	.cfi_startproc
	subq		$8, %rsp
	.cfi_def_cfa_offset 16
	movl		$_ZStL8__ioinit, %edi
	call		_ZNSt8ios_base4InitC1Ev
	movl		$__dso_handle, %edx
	movl		$_ZStL8__ioinit, %esi
	movl		$_ZNSt8ios_base4InitD1Ev, %edi
	addq		$8, %rsp
	.cfi_def_cfa_offset 8
	jmp		__cxa_atexit
	.cfi_endproc
.LFE1733:
	.size		_GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
	.section	.text.unlikely
.LCOLDE4:
	.section	.text.startup
.LHOTE4:
	.section	.init_array,"aw"
	.align 8
	.quad		_GLOBAL__sub_I_main
	.local		_ZStL8__ioinit
	.comm		_ZStL8__ioinit,1,1
	.section	.rodata.cst16,"aM",@progbits,16
	.align 16
.LC0:
	.long		1
	.long		2
	.long		3
	.long		4
	.align 16
.LC1:
	.long		5
	.long		5
	.long		6
	.long		7
	.hidden		__dso_handle
	.ident		"GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
	.section	.note.GNU-stack,"",@progbits
```


```asm
.LC0:
	.long	1
	.long	2
	.long	3
	.long	4
	.align 16
.LC1:
	.long	5
	.long	5
	.long	6
	.long	7
```