Designing costless abstractions

Bart Verhagen
bart@verhagenconsultancy.be

16 November 2019

What???

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

xkcd. *Is It Worth the Time?* URL: https://xkcd.com/1205/

Costless abstractions

Software abstraction

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.[1]

Costless

Having no cost[2]

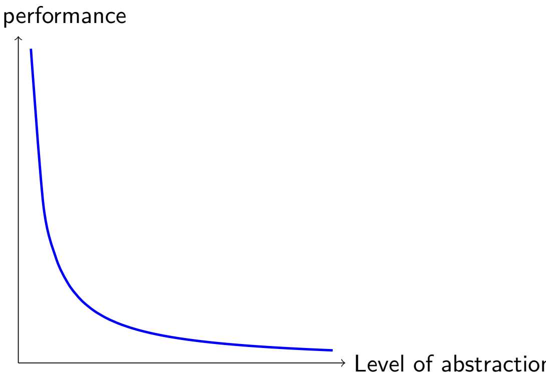
Costless abstractions or zero-overhead abstractions

An abstraction with no additional *runtime* cost compared to not using the abstraction

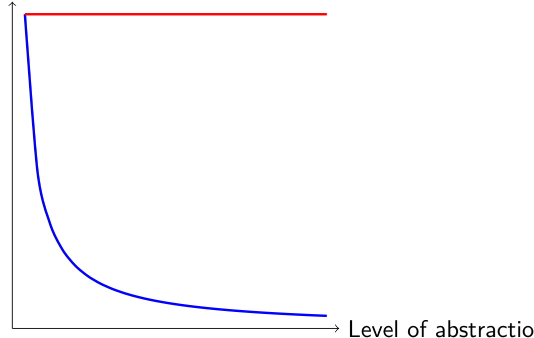
[1] John V. Guttag. *Introduction to Computation and Programming Using Python, Spring 2013 Edition*. MIT Press, 2013. ISBN: 9780262519632

[2] Wiktionary. *costless — Wiktionary, The Free Dictionary*. 2019. URL: https://en.wiktionary.org/w/index.php?title=costless

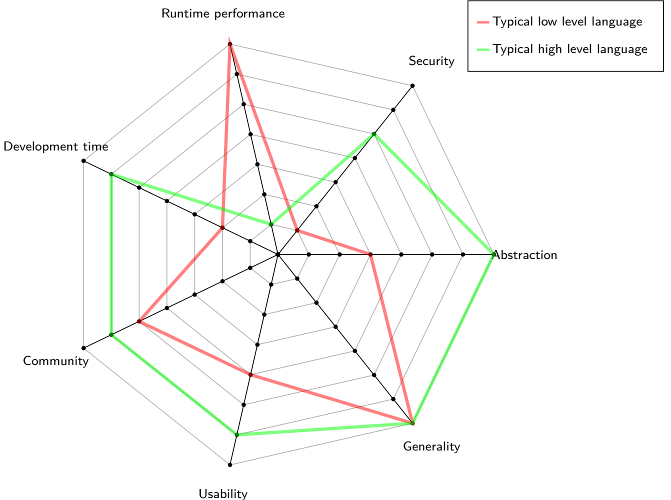
How we typically talk about it



Costless abstractions



How it actually is (significantly simplified)



University of Washington - Paul G. Allen School of Computer Science and Engineering. *Evaluating Programming Languages*. 2019. URL: https://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html

Costless abstractions: the holy grail?

std::unique_ptr

```
/**
 * Heavily simplified version of std::unique_ptr
 */
template<typename T>
class unique_ptr {
    public:
        explicit unique_ptr(T* resource) :
            m_resource(resource) {}

        unique_ptr(const unique_ptr<T>&) = delete;
        unique_ptr(unique_ptr<T>&&) noexcept = delete;

        ~unique_ptr() { delete m_resource; }

        unique_ptr& operator=(const unique_ptr<T>&)
            = delete;
        unique_ptr& operator=(unique_ptr<T>&&) noexcept
            = delete;
        // Other operators and functions omitted

    private:
        T* m_resource;
};

int main() {
    unique_ptr<int> i(new int());
    doSomethingThatUsesVariable(i);
}
```

Reference implementation

```cpp
int main() {
    auto i = new int();
    doSomethingThatUsesVariable(i);
    delete i;
}
```

```asm
; Compiled with x86-64 gcc 9.2, -O2
main:
    sub     rsp, 8
    mov     edi, 4
    call    operator new(unsigned long)
    mov     DWORD PTR [rax], 0
    mov     rdi, rax
    mov     esi, 4
    call    operator delete(void*, unsigned long)
    xor     eax, eax
    add     rsp, 8
    ret

; Compiled with clang 8.0.0, -O2
main:
    push    rax
    mov     edi, 4
    call    operator new(unsigned long)
    mov     dword ptr [rax], 0
    mov     qword ptr [rsp], rax
    mov     rdi, rax
    call    operator delete(void*)
    xor     eax, eax
    pop     rcx
    ret
```

std::unique_ptr

```cpp
/**
 * Heavily simplified version of std::unique_ptr
 */
template<typename T>
class unique_ptr {
    public:
        explicit unique_ptr(T* resource) :
            m_resource(resource) {}

        unique_ptr(const unique_ptr<T>&) = delete;
        unique_ptr(unique_ptr<T>&&) noexcept = delete;

        ~unique_ptr() { delete m_resource; }

        unique_ptr& operator=(const unique_ptr<T>&)
            = delete;
        unique_ptr& operator=(unique_ptr<T>&&) noexcept
            = delete;
        // Other operators and functions omitted

    private:
        T* m_resource;
};

int main() {
    unique_ptr<int> i(new int());
    doSomethingThatUsesVariable(i);
}
```

```asm
; Compiled with x86-64 gcc 9.2, -O2
main:
    sub     rsp, 24
    mov     edi, 4
    call    operator new(unsigned long)
    mov     DWORD PTR [rax], 0
    mov     rdi, rax
    mov     QWORD PTR [rsp+8], rax
    mov     esi, 4
    call    operator delete(void*, unsigned long)
    xor     eax, eax
    add     rsp, 24
    ret

; Compiled with clang 8.0.0, -O2
main:
    push    rax
    mov     edi, 4
    call    operator new(unsigned long)
    mov     dword ptr [rax], 0
    mov     rdi, rax
    call    operator delete(void*)
    xor     eax, eax
    pop     rcx
    ret
```

std::unique_ptr

manual memory management[1]

```
; Compiled with x86-64 gcc 9.2, -O2
main:
    sub     rsp, 8
    mov     edi, 4
    call    operator new(unsigned long)
    mov     DWORD PTR [rax], 0
    mov     rdi, rax
    mov     esi, 4
    call    operator delete(void*, unsigned long)
    xor     eax, eax
    add     rsp, 8
    ret

; Compiled with clang 8.0.0, -O2
main:
    push    rax
    mov     edi, 4
    call    operator new(unsigned long)
    mov     dword ptr [rax], 0
    mov     qword ptr [rsp], rax
    mov     rdi, rax
    call    operator delete(void*)
    xor     eax, eax
    pop     rcx
    ret
```

std::unique_ptr implementation[2]

```
; Compiled with x86-64 gcc 9.2, -O2
main:
    sub     rsp, 24
    mov     edi, 4
    call    operator new(unsigned long)
    mov     DWORD PTR [rax], 0
    mov     rdi, rax
    mov     QWORD PTR [rsp+8], rax
    mov     esi, 4
    call    operator delete(void*, unsigned long)
    xor     eax, eax
    add     rsp, 24
    ret

; Compiled with clang 8.0.0, -O2
main:
    push    rax
    mov     edi, 4
    call    operator new(unsigned long)
    mov     dword ptr [rax], 0
    mov     rdi, rax
    call    operator delete(void*)
    xor     eax, eax
    pop     rcx
    ret
```

[1] Matt Godbolt. *Compiler explorer.* URL: https://godbolt.org/z/-mdgmt

[2] Matt Godbolt. *Compiler explorer.* URL: https://godbolt.org/z/QwRwpF

Ensuring std::unique_ptr is costless

```
/**
 * Heavily simplified version of std::unique_ptr
 */
template<typename T>
class unique_ptr {
    public:
        explicit unique_ptr(T* resource) :
            m_resource(resource) {}

        unique_ptr(const unique_ptr<T>&) = delete;
        unique_ptr(unique_ptr<T>&&) noexcept = delete;

        ~unique_ptr() { delete m_resource; }

        unique_ptr& operator=(const unique_ptr<T>&)
            = delete;
        unique_ptr& operator=(unique_ptr<T>&&) noexcept
            = delete;
        // Other operators and functions omitted

    private:
        T* m_resource;
};

int main() {
    unique_ptr<int> i(new int());
    doSomethingThatUsesVariable(i);
}
```

The costless inlining law

A sufficiently smart compiler will *always* inline a function if - and only if - inlining is faster in *all* cases w.r.t. not inlining it.

CppCon 2019: There are no Zero-cost Abstractions

Unique_ptr costless: are we sure?

```
#include <memory>

int foo(std::unique_ptr<int> bar) noexcept;
```

Chandler Carruth. "There Are No Zero-cost Abstractions". In: CppCon, 2019

std::async

```cpp
#include <future>
#include <iostream>
#include <numeric>
#include <vector>

template <typename RandomIt>
int parallel_sum(RandomIt beg, RandomIt end) {
    constexpr unsigned int magicalTurningPoint = 100U;
    auto len = end - beg;
    if (len < magicalTurningPoint) { return std::accumulate(beg, end, 0); }

    RandomIt mid = beg + len/2;
    auto handle = std::async(std::launch::async, parallel_sum<RandomIt>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main() {
    constexpr int nbOfValues = 10000;
    std::vector<int> v(nbOfValues, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << std::endl;
}
```

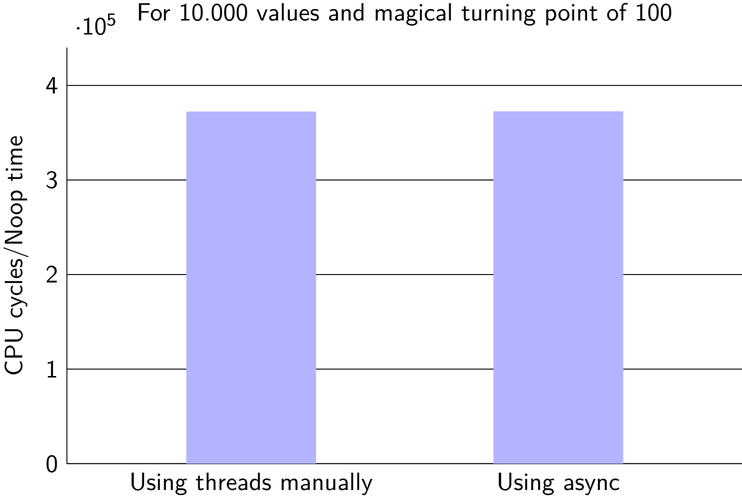
```
The sum is 10000
```

cppreference.com. *std::async*. URL: https://en.cppreference.com/w/cpp/thread/async

std::async (conceptually)

```cpp
#include <thread>
template<typename Result>
class future {
    public:
        template<class Function, class... Args>
        explicit future(Function&& f, Args&&... args ) {
            m_thread = std::thread([this, f, args...]() { m_result = f(args...); });
        };

        future(const future<Result>&) = delete;
        ~future() = default;
        future<Result>& operator=(const future<Result>&) = delete;

        void wait() /*const*/ { m_thread.join(); }
        Result get() {
            wait();
            return m_result;
        }

    private:
        std::thread m_thread;
        Result m_result;
};

template< class Function, class... Args>
future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...> >
async(Function&& f, Args&&... args) { // std::async implementation of the std::launch::async policy
    return future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>(f, args...);
}
```

std::async measurements



Quick C++ Benchmark. *Compare async vs threads.* URL: http://quick-bench.com/-j83zmDXSqOhFBEQwkbQRjhp8uA

An unbloated, type-safe stack

```
#include <iostream>
#include <memory>
#include <vector>

namespace detail {
    class Stack {
        public:
            Stack() = default;
            std::shared_ptr<void> top();
            void push(std::shared_ptr<void> element);
            void pop();

        private:
            std::vector<std::shared_ptr<void> > m_stack;  // Choose the backend of your liking
    };
} // namespace detail

template<typename T> class Stack {
    public:
        void push(std::shared_ptr<T> element) { m_stack.push(element); }
        std::shared_ptr<T> top() { return std::static_pointer_cast<T>(m_stack.top()); }
        void pop() { m_stack.pop(); }

    private:
        detail::Stack m_stack;
};
```

Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley Professional, May 2005. ISBN: 9780321334879

Thin wrappers

Policies

```
inline constexpr std::execution::sequenced_policy seq { /* unspecified */ };
inline constexpr std::execution::parallel_policy par { /* unspecified */ };
inline constexpr std::execution::parallel_unsequenced_policy par_unseq { /* unspecified */ };
inline constexpr std::execution::unsequenced_policy unseq { /* unspecified */ };
```

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
            class ForwardIt3, class BinaryOperation >
ForwardIt3 transform( ExecutionPolicy&& policy, ForwardIt1 first1, ForwardIt1 last1,
                      ForwardIt2 first2, ForwardIt3 d_first, BinaryOperation binary_op );
```

```
#include <algorithm>
#include <cstdlib>
#include <execution>

int main() {
    std::string s("hello");
    std::transform(std::execution::seq, s.begin(), s.end(), s.begin(),
                   [](unsigned char c) -> unsigned char { return std::toupper(c); });
    return EXIT_SUCCESS;
}
```

Khronos Group. *SYCL Parallel STL*. URL: https://github.com/KhronosGroup/SyclParallelSTL

Policies: straightforward implementation

```
namespace {
    enum class Policy {
        policy1,
        policy2
    };

    int executePolicy1() {
        return 1;
    }

    int executePolicy2() {
        return 2;
    }

    int executePolicy(Policy policy) {
        switch(policy) {
            case Policy::policy1:
                return executePolicy1();
            case Policy::policy2:
                return executePolicy2();
        }
    }
} // namespace

int main() {
    return executePolicy(Policy::policy2);
}
```

```
$ echo $?
2
```

Policies: assembler

Direct call[1]

```
; Compiled with x86-64 gcc 9.2 and clang 9.0.0, -O2
main:
    mov     eax, 2
    ret
```

Call using policies[2]

```
; Compiled with x86-64 gcc 9.2 and clang 9.0.0, -O2
main:
    mov     eax, 2
    ret
```

[1] Matt Godbolt. *Compiler explorer*. URL: https://godbolt.org/z/pM5qDh

[2] Matt Godbolt. *Compiler explorer*. URL: https://godbolt.org/z/1xORco

Policies: library implementation

```cpp
namespace {
    class Policy1 {};
    class Policy2 {};
    constexpr Policy1 policy1;
    constexpr Policy2 policy2;

    template<typename Policy> struct Execute {
        constexpr static int exec() {
            static_assert(sizeof(Policy) != sizeof(Policy),
                    "Please define a specialization for the given policy");
            return 0;
        }
    };

    template<> struct Execute<Policy1> {
        constexpr static int exec() { return 1; }
    };

    template<> struct Execute<Policy2> {
        constexpr static int exec() { return 2; }
    };

    template<typename Policy> int executePolicy(Policy) {
        return Execute<Policy>::exec();
    }
} // namespace

int main() {
    return executePolicy(policy2);
}
```

Partial/full template specialization

```cpp
#include <cfloat>
#include <climits>
#include <iostream>

template<typename T> class numeric_limits {
    public:
        static int min() {
            static_assert(sizeof(T) != sizeof(T), "The specialization for T does not exist");
            return 0;
        }
};

template<> class numeric_limits<int> {
    public:
        constexpr static bool is_integer = true;
        constexpr static bool is_signed = true;
        constexpr inline static int min() { return INT_MIN; }
};

template<> class numeric_limits<float> {
    public:
        constexpr static bool is_integer = false;
        constexpr static bool is_signed = true;
        constexpr inline static float min() { return FLT_MIN; }
};

int main() {
    std::cout << numeric_limits<int>::min() << std::endl;
    std::cout << numeric_limits<float>::is_integer << std::endl;
}
```

Iterators and iterator arguments

```
template< class RandomIt >
void sort(RandomIt first, RandomIt last);

template< class OutputIt, class Size, class Generator >
OutputIt generate_n( OutputIt first, Size count, Generator g );

template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
    std::vector<int> s;
    constexpr int size = 10;
    int counter = size;
    std::generate_n(std::back_inserter(s), size, [&counter]() { return --counter; });
    std::sort(s.begin() + 3, s.end() - 2);

    std::for_each(s.begin(), s.end(), [](auto s) {
        std::cout << s << " ";
    });
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

```
9 8 7 2 3 4 5 6 1 0
```

Ranges (C++20)

```
#include <range/v3/all.hpp>

using namespace ranges::v3;

int main() {
    return accumulate(ranges::view::iota(1)
              | view::transform([] (int x) { return x * x;})
              | view::remove_if([](int i){ return i % 2 == 1; })
              | view::take(10), 0);
}
```

```
$ echo $?
1540
```

Eric Niebler. "N4128: Ranges for the Standard Library, Revision 1". In: *Standard C++ Foundation* (Oct. 2014). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html

Ranges library. URL: https://en.cppreference.com/w/cpp/ranges

Are ranges costless?

```cpp
#include <range/v3/all.hpp>

using namespace ranges::v3;

int main() {
    return accumulate(ranges::view::iota(1)
             | view::transform([] (int x) { return x * x;})
             | view::remove_if([](int i){ return i % 2 == 1; })
             | view::take(10), 0);
}
```

```asm
; Compiles with clang 9.0.0, -O2
main:
    mov     eax, 1540
    ret
```

Matt Godbolt. *Compiler explorer*. URL: https://godbolt.org/z/LD5XvN

Eric Niebler. *Range Comprehensions*. 2014. URL: http://ericniebler.com/2014/04/27/range-comprehensions/

Ranges: the concept

```
template<typename T>
auto make_pipeline(T element) {
    return Pipeline<T>(element);
}

template<typename Outer, typename T1, typename T2>
auto operator|(Pipeline<T1, T2> inner, Outer outer) noexcept {
    return Pipeline<Outer, Pipeline<T1, T2»(outer, inner);
}

template<typename Outer, typename Inner>
auto operator|(Pipeline<Inner> inner, Outer outer) noexcept {
    return Pipeline<Outer, Pipeline<Inner»(outer, inner);
}

int main() {
    auto iota = make_pipeline([]() { static int x = 1; return x++;});
    auto transform = [](int x) { return x * x; };

    auto pipeline = iota | transform;

    int sum = 0;
    for(int i = 0; i < 10; ++i) {
        sum += pipeline();
    }
    return sum;
}
```

Ranges: the concept

```
template<typename... T> class Pipeline {
    public:
        Pipeline() = default;
        template<typename ElementType> auto operator() (ElementType x) {
            return Pipeline<T...>::operator()(x);
        }
};

template<typename T, typename... T2> class Pipeline<T, T2...> {
        T m_t1;
        Pipeline<T2...> m_inner;
    public:
        Pipeline(T t1, T2... t2) : m_t1(t1), m_inner(t2...) { }
        Pipeline(int begin, int end, T t1, T2... t2) : m_t1(t1), m_inner(t2...) { }

        template<typename... ElementType> auto operator() (ElementType... x) {
            return m_t1(m_inner(x...));
        }
};

template<typename T> class Pipeline<T> {
        T m_t1;
    public:
        Pipeline(T t1) : m_t1(t1) {}
        Pipeline(int begin, int end, T t1) : m_t1(t1) {}

        template<typename... ElementType> auto operator() (const ElementType... x) {
            return m_t1(x...);
        }
};
```

Conclusion

xkcd. *Abstraction*. URL: https://xkcd.com/676/

Thank you!

?