

Hiding your Implementation Details is Not So Simple

Amir Kirsh

Cpp
N  **rth**
2024

(picture by Amir Kirsh, no AI involved)

About me

Lecturer

Academic College of Tel-Aviv-Yaffo
Tel-Aviv University

Member of the Israeli ISO C++ NB

Co-Organizer of the **CoreCpp**
conference and meetup group



Trainer and Advisor
(C++, but not only)



Hiding your Implementation Details

Hiding your Implementation Details

Why?

Encapsulation

Protect Object Integrity

Expose the necessary interfaces only.

Easier to Use

Users see only what they need to.

Improves Maintainability

Internal changes don't affect external code.

Easier to Debug

Data modifications happens internally, in specific places.

Decoupling

Reduces Dependencies

Components depend only what we actually expose.

Facilitates Changes

Changes in one part of the system don't necessarily affect others.

Enhances Reusability

Components become more generic, thus can be better reused.

Improves Testability

Easier to test components in isolation.

Hiding your Implementation Details *is Not So Simple*

Hiding your Implementation Details *is Not So Simple*

And it's not only me saying that...

David L. Parnas
Modularization by Information Hiding

Information distribution aspects of design methodology, 1971

On the criteria to be used in decomposing systems into modules, 1972

David L. Parnas

Modularization by Information Hiding

[Information distribution aspects of design methodology, 1971](#)

[On the criteria to be used in decomposing systems into modules, 1972](#)

(And also, on another note: [Software Aging, 1994](#))

Information hiding is harder than it looks

Software engineering educational standards are much lower than those for other engineering fields, there are other reasons for failing to apply information hiding that I have observed.

1. The Flowchart Instinct
2. Planning for Change Seems like too much Planning
3. Bad Models
4. Extending Bad Software
5. Assumptions often go unnoticed:
Major Flaw in my KWIC Index
6. Reflecting the system environment in the software structure.
7. "Oh, too bad we changed it"

Hiding your Implementation Details *is Not So Simple*

Why?

1. Because we're lazy and don't think it's that important*

“Lazy”, may include:

Time pressure: we have to cut the “fancy design” somewhere.


We don't think it's that important, may include:

We think we may change this later easily, if required
(Reality: in many cases you can't, not easily, if at all).

1. Because we're lazy and don't think it's that important

```
std::pair<First, Second> p;  
auto k = p.first;  
auto v = p.second;
```

Yes, there are other ways to extract first and second, but this one is still available and exposes private fields



I will try to convince you in this talk:

- Not to be lazy
- That it is important, even when it doesn't seem to be

2. Because we do not realize we expose private details

```
class Foo {  
    [...]  
public:  
    const std::map<string, int>& getProperties() const;  
};
```




What are the private details
exposed?

3. Because of technical constraints

```
class Foo {  
public:  
    // we prefer the ctor below to be private but it doesn't compile :(  
    Foo(int value) : value(value) {}  
    static unique_ptr<Foo> create(int value) {  
        return std::make_unique<Foo>(value);  
    }  
};  
  
auto foo = Foo::create(7);
```

Wait, don't jump, there are solutions
for this, we will discuss it later



⇒ **Technical constraints can be addressed and resolved.**

(Code: <https://compiler-explorer.com/z/Mvn9h3G88>)

4. Because there are tradeoffs

- We may want to avoid solutions relying on interfaces, to avoid virtual calls
- Exposing data to the user may be valuable for performance, e.g. exposing that we use a random access container, accessible with an index

Tradeoffs should be discussed:

- In some cases the achieved goal doesn't justify the design compromise.
- There may be a solution that meets both needs.

Our Goal

Investigate code examples:

- See how we expose implementation details
- Understand why it's not the best design, per case
- Propose a better design, hiding the implementation details

Let's start!

std::pair

std::pair

not hiding your privates is wrong

Data members should be private

`std::pair.first, std::pair.second` => is a *language accident...*

Why?

Because it doesn't properly allow different behaviors, e.g. a pair initialized with a single number, with the second being lazy evaluated to its square

std::pair

```
// assume function is constrained on PAIR having fields first and second
template<typename PAIR>
std::ostream& operator<<(std::ostream& out, const PAIR& pair) {
    return out << pair.first << ' ' << pair.second;
}

// ok
auto stdpair = std::make_pair("bye", 42);
std::cout << stdpair << std::endl;

// can we allow our pair to lazily evaluate its second?
auto squarepair = SquarePair(7);
std::cout << squarepair << std::endl;
```

(doable but not straightforward: <http://coliru.stacked-crooked.com/a/4c31320c394bcbb5>)

structs, in general

structs, in general

```
struct Order {  
    Price price;  
    Quantity quantity;  
};
```



What's the problem?

Supporting Generic Orders


Supporting Generic Orders

```
class Order {  
    Price price;  
    Quantity quantity;  
public:  
    // [ctor]  
    Price getPrice() const {  
        return price;  
    }  
    Quantity getQuantity() const {  
        return quantity;  
    }  
};
```


Supporting Generic Orders

```
class TimedOrder {  
    std::vector<chrono::time_point, Price> price_limits;  
    Quantity quantity;  
public:  
    // [ctor]  
    Price getPrice() const {  
        // fetch the proper price  
    }  
    Quantity getQuantity() const {  
        return quantity;  
    }  
};
```

We can treat this as a regular order, based on “Duck Typing”.



We rely here on “Static Polymorphism”.
But it may work also with “Dynamic Polymorphism”.



But if it's just a const member?

Just a const member...


```
class Person {  
    string name;  
public:  
    const long id;  
    Person(string name, long id)  
        : name(std::move(name)), id(id) {}  
};
```

What's the problem?



Just a const member...

```
class Person {  
    string name;  
public:  
    const Id id;  
    Person(string name, Id id)  
        : name(std::move(name)), id(id) {}  
};
```



Requirements tend to change:
A person may be identified by
SSN or Passport number.
The person shall hold an
internal Id object that would hold
either an SSN number or
passport details.

(What's the problem, still?)

Just a const member...


```
class Person {  
    string name;  
public:  
    const Id id;  
    Person(string name, Id id)  
        : name(std::move(name)), id(id) {}  
};
```

Requirements continue to change...



Just a const member - ... not any more

```
class Person {  
    string name;  
public:  
    const Id id;  
    Person(string name, Id id)  
        : name(std::move(name)), id(id) {}  
};
```



Requirements continue to change:
Passport number may change, when
issuing a new passport.
Person class shall allow that, keeping
the old passport numbers along with
the new one.

Person - as it should be to begin with

```
class Person {  
    string name;  
    Id id;  
public:  
    Person(string name, Id id)  
        : name(std::move(name)), id(id) {}  
    // expose relevant Id inquiries + relevant setters  
    // (e.g. newPassport(PassportDetails) etc.)  
};
```

API (including internal API of all kinds!)

Hyrum's Law

(“Software engineering at Google: lessons learned from programming over time”):

- Developers come to depend on all observable traits and behaviors of an interface, even if they are not defined in the contract.
- With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

API

Studies show that applications tend to use a small part of an API.

API

Studies show that applications tend to use a small part of an API.

Unused API is much more bug prone!

API

Studies show that applications tend to use a small part of an API.

Unused API is much more bug prone!

When there are too many options (e.g. too many overloads), a wrong option would be selected at some point.

API

Studies show that applications tend to use a small part of an API.

Unused API is much more bug prone!

When there are too many options (e.g. too many overloads), a wrong option would be selected at some point.

Less options means also less required tests.

Be Lean and Mean

1. Keep your API short and concise.
(add as needed, and only for there is a real need).
2. Limit your API to what should actually be used
(arguments, return value etc.) - do not expose too much of your design!
3. Different usages may get different *view* (or copy) of the same data, exposing only what is relevant.

It reduces the programmer's cognitive load!



Beginner's mind, expert's mind -
How we think about, read, write
and learn to code

Dawid Zalewski

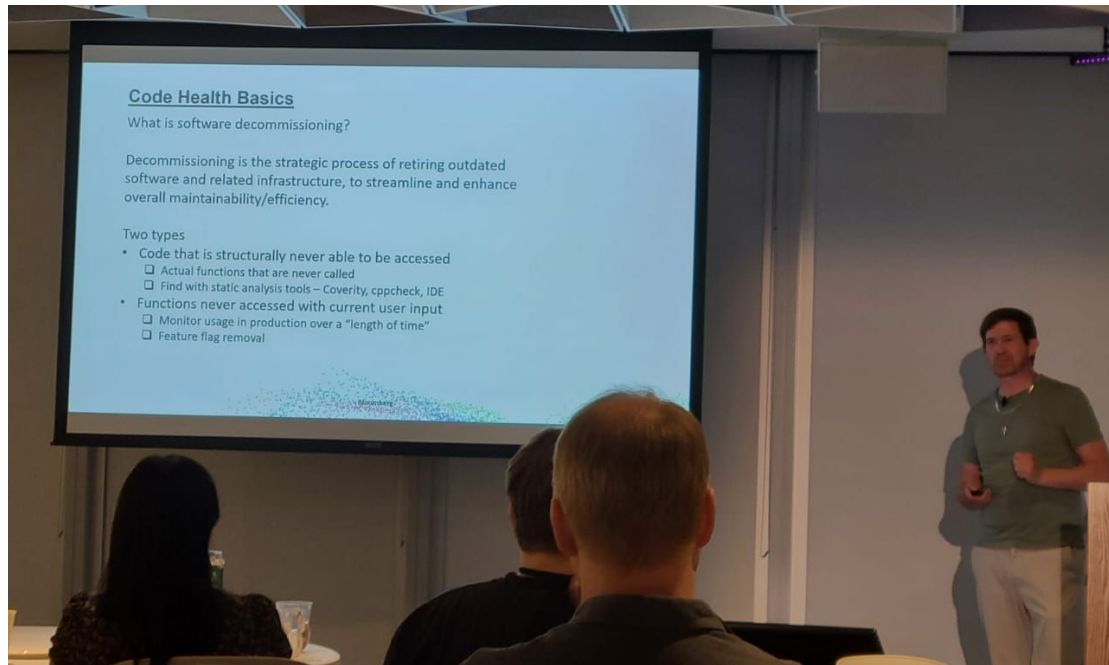


Less code to decommission!

Software Engineering

Completeness : Knowing when you are done and why it matters

Peter Muldoon



We keep the 'O' in SOLID

3 Ideas of how to **Modify Software (2/2)**

1. Just edit the source code (Editing Code)
2. Edit source with discipline (Editing Code)
3. Live Patching (Editing Code)

Think to yourself if these strategies meet the below goal:

1. **Develop software with a 'solid and stable core' (like hardware)**
2. Maintain flexibility and ease at which we can extend software

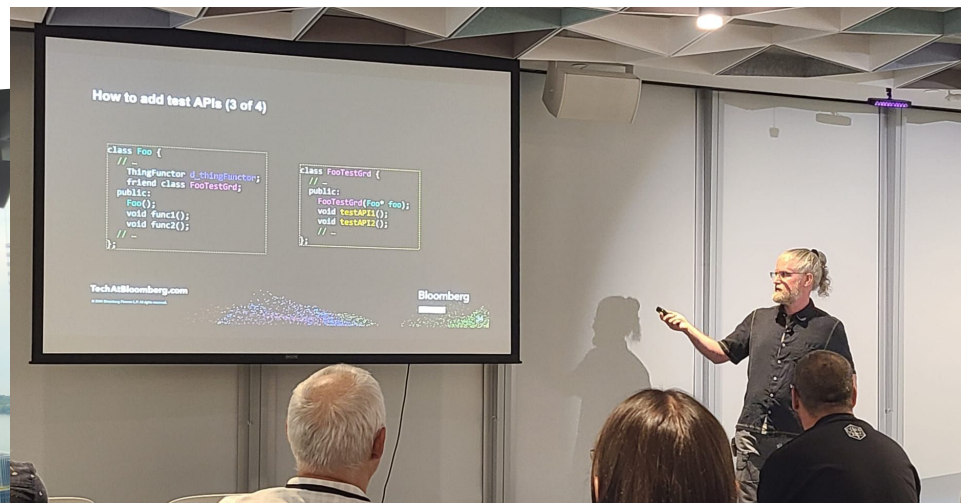
37



A Study of Plugin Architectures for Supporting Extensible Software

Mike Shah

And make our code easier to test!



Testability and API Design

John Pavan • Aram Chung • Lukas Zhao

But: it's context specific!

It's context specific!

The interface that we provide shall be as narrow as possible.

But something that is reasonably exposed in a certain context may need to be considered “private” in another context!

- Should we expose different behavior for different contexts?
- Can we identify the different contexts?

Tony Van Eerd's Koan - The Master and his Dog

Tony Van Eerd's Koan - The Master and his Dog



Tony Van Eerd

@tvaneerd

...

Master, I am honoured by your visit.

I was out walking my dog. Your function, why does it take
BigCommonStruct instead of just x and y?

BCS is where we keep x and y, master.

Bark

He's hungry.

Shall I prepare him food?

No, just open the fridge, let him take what he wants.

[#CppKoan](#)

9:56 PM · Mar 23, 2018

The Master and his Dog - Act II

The Master and his Dog - Act II



Amir Kirsh
@AmirKirsh

Master, tho I trust your dog completely, I still got a concern.
What's your concern? the fridge engine is in the private!

It's context specific!

Something may be required to be public for one usage.
Yet, have no reason to be public for other usages.

It's context specific!

Something may be required to be public for one usage.

Yet, have no reason to be public for other usages.

Because:

- If they don't need it, don't give it.
- If they get it, they'll use it.
- If they use it, they'll abuse it.
- Trust no one!

It's context specific!

Something may be required to be public for one usage.

Yet, have no reason to be public for other usages.

Because:

- If they don't need it, don't give it.
- If they get it, they'll use it.
- If they use it, they'll abuse it.
- Trust no one!



Having “things” in the private is not enough!

Passing args, exposing only what's needed

How can we do that?

Passing args, exposing only what's needed

Language tools:

- Value semantics: send values, if needed wrap in a class
- Interfaces
- Pimpl idiom
- Wrapper classes as a view, wrapping the original
- C++20 Concepts (?)

Exercise - “Const Map, Mutable Vals”

Initialize a map, then pass it in a way such that values can be modified but the map itself cannot be modified.

“Const Map, Mutable Vals” - Concept Approach

A Meander

Meandering Through C++
to Create ranges::to

Rudyard Merriam



A Meander: Concepts

What's a Concept?

- A constraint on template arguments
- Evaluated to a boolean value, at compile time
- May be used for overload resolution



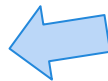
A much more elegant and
simple substitute for SFINAE

First Example

We want to overload two generic functions:

```
// for types that we can iterate over
template<typename Iterable>
void print(const Iterable& iterable) {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}

// for all other types
template<typename T> void print(const T& t) {
    cout << t << endl;
}
```



current code
doesn't work,
why?

First Example - SFINAE as a rescue

We want to overload two generic functions:

```
// for types that we can iterate over
template<typename Iterable, std::enable_if_t<is_iterable_v<Iterable>>*> dummy = nullptr>
void print(const Iterable& iterable) {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}

// for all other types
template<typename T, std::enable_if_t<!(is_iterable_v<T>)>*> dummy = nullptr>
void print(const T& t) {
    cout << t << endl;
}
```

note that `is_iterable_v` is not a language type trait, it is defined inside our code

works...

but



Let's do it with concepts

Let's do it with concepts

We will use an existing concept defined by C++20, called

range

(Yes, it relates to `std::ranges` that we will discuss later today)

It's not a complex concept, see:

<https://en.cppreference.com/w/cpp/ranges/range>

1: Requiring a constraint

```
// for types that we can iterate over
template<typename T> requires std::ranges::range<T>
void print(const T& iterable) {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}
```



[Code](#)

```
// for all other types
template<typename T> void print(const T& t) {
    cout << t << endl;
}
```

1: Requiring a constraint - Option B

```
// for types that we can iterate over
template<typename T>
void print(const T& iterable) requires std::ranges::range<T> {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}

// for all other types
template<typename T> void print(const T& t) {
    cout << t << endl;
}
```



Code

OK, not better than previous version, but sometimes required

2: Constraining the template type

```
// for types that we can iterate over
template<std::ranges::range T>
void print(const T& iterable) {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}
```



[Code](#)

```
// for all other types
template<typename T> void print(const T& t) {
    cout << t << endl;
}
```

3: Constraining the function argument

```
// for types that we can iterate over
void print(const std::ranges::range auto& iterable) {
    for(const auto& v: iterable) {
        cout << v << endl;
    }
}
```

```
// for all other types
void print(const auto& t) {
    cout << t << endl;
}
```



[Code](#)

C++20 allows *auto* on function arguments, making the function a template function, this is called "abbreviated function template"

Creating our own concept

Creating our own concept

```
template<typename T>
concept Meowable = requires(const T t) {
    t.meow(); // t has a method meow that should be const
};

struct Cat {
    void meow() const { cout << "meow\n"; }
};

// using a concept to constraint a function
void do_meow(const Meowable auto& meowable) {
    meowable.meow();
}
```

Code: <https://compiler-explorer.com/z/YnnzKfhYn>

The *requires* keyword

The *requires* keyword is used for two different usages:

- [requires clause](#)

Specifying constraints on a template argument or on a function declaration, e.g.: `template<typename T> requires Addable<T> // ...`

- [requires expression](#)

Describing a constraint, e.g.:

```
template<class T>
concept Fooable = requires (T t) {
    t.foo();
};
```

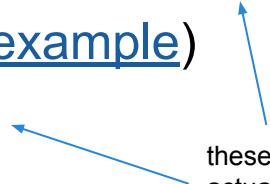
requires clause

template<typename T> *requires* ...

Must be followed a constant expression that can be evaluated to a bool:

- A constexpr bool (e.g. *true* / *false* - see [naive code](#), or a [better example](#))
- A bool constant expression (e.g. *(sizeof(T) >= 4)* - see [example](#))
- A bool constant expression based on type trait ([example](#))
- A concept ([example](#))
- An ad-hoc concept (requires “requires requires” - [example](#))
- A conjunction of any of the above ([example](#))

these are
actually
based on the
same idea



requires expression

```
template<class T> concept Fooable = requires (T t) { t.foo(); };
```

Used to express constraints for a concept.

(But, not the only way to define a concept, we will shortly see other ways).


Defining a concept (1)

```
template<class T> concept Fooable = <constraint>;
```

The *constraint* can be expressed as:

- A bool constant expression ([example](#))
- A bool constant expression based on other concepts ([example](#))
- A *requires* expression, e.g.:
template<class T> concept Fooable = *requires* <requirements>
 - see next slide
- A conjunction of any of the above ([example](#))

this example
is also a
conjunction



Defining a concept (2)

```
template<class T> concept Fooable = requires <requirements>;
```

With *requirements* being expressed as:

- Curly body for requirements, without arguments ([example](#))
- Argument list + curly body for requirements
([example](#) - [alternative example with declval, just for exposition](#))



**arguments are
unevaluated!**

Defining a concept (3)

Inside the concept's body, we can have:

1. Unevaluated expressions that just need to be valid (i.e. can compile).
2. Requirements, using an internal *requires* clause.
3. Curly braces on an expression, to be injected into another concept.
4. A requirement for the existence of an internal type, with the `typename` keyword.

```
template<class T>
concept AllSortOfChecks = requires {
    T::foo(); // 1
    requires T::Size == 2; // 2 -- the requires keyword is required!
    {T::bar()} -> std::same_as<int>; // 3
    typename T::inner_type; // 4 -- the typename keyword is required!
};
```

Code: <https://compiler-explorer.com/z/hqedcPPvr>

The concept's param (1)

A concept is always templated over at least one template parameter.

When using the concept, the first template argument may be automatically injected, in cases where the concept refers directly to a type, e.g.:

```
// 1. concept on template argument: concept's 1st argument is injected
//    by the compiler, as if we wrote: Printable<T>
template<Printable T>
void print1(const T& t) { cout << t << endl; }

// 2. concept on auto parameter: concept's 1st argument is injected
//    by the compiler, as if we wrote: Printable<decltype(t)>
void print2(const Printable auto& t) { cout << t << endl; }
```

The concept's param (2)

In cases where the concept is not directly referring to a type, the concept's parameter is not injected and thus shall be manually provided:

```
// examples where concept's 1st argument is not injected
// and shall be provided explicitly:
// [1]
template<typename T> requires Printable<T>
void print(const T& t) { cout << t << endl; }
// [2]
if constexpr(Printable<T>) { ... }
// [3]
static_assert(Printable<int>); // OK, int is Printable
```

concepts with more than one param

A concept may have more than one parameter, the first parameter may be injected, as seen before, the rest shall be provided.

Example:

```
template<typename T>
concept Dereferenceable = requires(T t) {
    *t;
};
```

```
template<typename T, typename ValueType>
concept DereferenceableTo =
    Dereferenceable<T> &&
    std::same_as<std::remove_cvref_t<decltype(*std::declval<T>())>, ValueType>;
```



This concept has two params
usage appears on next slide

concepts with more than one param

Usage example:

```
void print(const auto& v) {  
    cout << "Simple value: " << v << endl;  
}  
  
void print(const Dereferenceable auto& t) {  
    cout << "Dereferenceable, inner value: " << *t << ", at: " << t << endl;  
}  
  
void print(const DereferenceableTo<char> auto& t) {  
    cout << "Dereferenceable to char: " << t << ", at: " << (void*)t << endl;  
}
```



This is the one with two params

Code: <https://compiler-explorer.com/z/q6GbjTjb9>

Exercise



Implement a concept for “Twople<First, Second>” so we can write the following code:

```
void foo(const Twople<int, double> auto&) {  
    cout << "Twople<int, double>\n";  
}
```

Above function shall be able to accept both *std::pair<int, double>* and *std::tuple<int, double>*

Skeleton: <https://compiler-explorer.com/z/7qc4WoK3r>

Solution (don't peek): <https://compiler-explorer.com/z/YjKGrevK6>

Exercise



Implement a concept for “OneOf” so we can write the following two overload functions:

```
void foo(const OneOf<char, int, long> auto&) { ... }
```

```
void foo(const OneOf<double, float> auto&) { ... }
```

Note that *OneOf* can take any amount of types!

Skeleton: <https://compiler-explorer.com/z/qeeYE88q5>

Solution (don't peek): <https://compiler-explorer.com/z/nWbdPj8eM>

concepts with non-type param(s)

The first parameter of a concept must be a type (think: why?)

The rest of the params can be non-types.

Example:

```
template<class T, size_t MIN_SIZE, size_t MAX_SIZE>  
concept SizeBetween = sizeof(T) >= MIN_SIZE && sizeof(T) <= MAX_SIZE;
```

Code: <https://compiler-explorer.com/z/h7nrvazvc>

Exercise



Implement a concept for “`TupleOf<Num_Elements>`” so we can write the following functions:

```
void foo(const TupleOf<2> auto&) { ... }
```

```
void foo(const TupleOf<3> auto&) { ... }
```

⇒ `foo(td::tuple{1, "two"});` – Should call the first one

⇒ `foo(td::tuple{1, "two", 3});` – Should call the 2nd one

Skeleton: <https://compiler-explorer.com/z/K4jvb8saj>

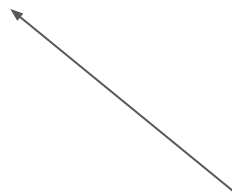
Note that [this](#) doesn't work

Solution (don't peek): <https://compiler-explorer.com/z/Y84hff1Eq>

A more advanced solution (don't peek): <https://compiler-explorer.com/z/Ed36e8T1f>

Exercise - “Const Map, Mutable Vals”


Initialize a map, then pass it in a way such that values can be modified but the map itself cannot be modified.



Let's try doing it with a concept

“Const Map, Mutable Vals” - Concept Attempt 1


```
// note: function may modify values but should not alert the map itself!  
void foo(ConstDictMutableVals auto& d) {  
    d[1] = 3;  
}  
  
int main() {  
    map<int, int> myMap = {{1, 0}, {2, 0}};  
    foo(myMap);  
    for (const auto& [k, v] : myMap) {  
        std::cout << k << ": " << v << std::endl;  
    }  
}
```



a concept, requiring only
the allowed operations

“Const Map, Mutable Vals” - Concept Attempt 2

```
// note: function may modify values but should not alert the map itself!  
void foo(ConstDictMutableVals auto& d) {  
    auto itr = d.find(1);  
    if(itr != d.end()) itr->second = 3;  
}  
  
int main() {  
    map<int, int> myMap = {{1, 0}, {2, 0}};  
    foo(myMap);  
    for (const auto& [k, v] : myMap) {  
        std::cout << k << ": " << v << std::endl;  
    }  
}
```



a concept, requiring only the allowed operations (which should not include the [] operator!).

ConstDictMutableVals - The Concept

```
template <typename T>
concept ConstDictMutableVals = requires(T t, typename T::key_type key) {
    typename T::iterator;
    typename T::key_type;
    typename T::mapped_type;
    { t.find(key) } -> std::same_as<typename T::iterator>;
    t.find(key) != t.end();
};
```

Code: <https://coliru.stacked-crooked.com/a/955910376d3152b4>

“Const Map, Mutable Vals” - Wrapper Approach

“Const Map, Mutable Vals” - Wrapper Approach

// note: function may modify values but cannot alert the map itself!

```
template<Dictionary Dict>
```

```
void foo(ConstDictMutableValues<Dict> d) {
```

```
    d.assign(1, 3);
```

```
}
```

```
int main() {
```

```
    map<int, int> myMap = {{1, 0}, {2, 0}};
```

```
    foo(ConstDictMutableValues(myMap));
```

```
    for (const auto& [k, v] : myMap) {
```

```
        std::cout << k << ": " << v << std::endl;
```

```
    }
```

```
}
```

the underlying container
(using a concept to restrict it).

a wrapper class exposing
only what should be exposed
(see next slide)

ConstDictMutableValues - The Wrapper

```
template<Dictionary Dict>
class ConstDictMutableValues {
    Dict& d_;
public:
    explicit ConstDictMutableValues(Dict& d) : d_(d) {}
    bool assign(const Key& key, Value value) {
        auto itr = d_.find(key);
        if(itr == d_.end()) return false;
        itr->second = std::move(value);
        return true;
    }
};
```



This is the wrapper class,
providing only the required
functionality

Code: <https://coliru.stacked-crooked.com/a/163f4dc6e2974f77>

Hiding Smart Pointers and Hierarchies

Math Expression Hierarchy

// we want the following code:

```
Expression* e = new Sum (  
    new Exp(new Number(3), new Number(2)),  
    new Number(-1)  
);  
cout << *e << " = " << e->eval() << endl;  
delete e;
```

// to print something like:

// ((3 ^ 2) + (-1)) = 8

But..., do you like it?

// what is bothering you with the code below?

```
Expression* e = new Sum (  
    new Exp(new Number(3), new Number(2)),  
    new Number(-1)  
);  
cout << *e << " = " << e->eval() << endl;  
delete e;
```

 Non-symmetric calls to new and delete

Why do we use new and delete to begin with?
Shouldn't we use smart pointers? 🤔

Let's move to smart pointers!

// what is bothering you with the code below?

```
auto e = make_unique<Sum>(  
    make_unique<Exp>(  
        make_unique<Number>(3),  
        make_unique<Number>(2)  
    ),  
    make_unique<Number>(-1)  
);  
cout << *e << " = " << e->eval() << endl;
```

Let's try to hide the `unique_ptr`

We aim for something like this

```
auto e = Sum(Exp(Number(3), Number(2)), Number(-1));  
cout << e << " = " << e.eval() << endl;
```

Why is it better?

```
auto e = Sum(Exp(Number(3), Number(2)), Number(-1));  
cout << e << " = " << e.eval() << endl;
```

// what makes the code above better? compared to:

```
auto e = make_unique<Sum>(
    make_unique<Exp>(make_unique<Number>(3), make_unique<Number>(2)),
    make_unique<Number>(-1)
);  
cout << *e << " = " << e->eval() << endl;
```

Hiding the use of smart pointers

```
class BinaryExpression: public Expression {
    unique_ptr<Expression> e1, e2;
public:
    template<typename Expression1, typename Expression2>
    BinaryExpression(Expression1 e1, Expression2 e2)
        : e1(make_unique<Expression1>(std::move(e1))),
          e2(make_unique<Expression2>(std::move(e2))) {}
    // ...
};
```

Hiding your implementation details!

The user doesn't have to know we use `unique_ptr`

We may want to change it later

It's a “private implementation detail”

And it's noisy

Code: <http://coliru.stacked-crooked.com/a/35f49a5a014224f8>

Talk: [Six Ways for Implementing Math Expressions Calculator in C++ - Amir Kirsh - CppCon 2023](#)

Related: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3086r2.pdf> - a proposal for a proxy class for representing type-erased pointers (though, use of such a proxy should also be hidden).

Hiding Inheritance Hierarchies Details

Using:

- **State Pattern**

(Employee is an Employee, regardless of his employment type - a state)

- **Strategy Pattern**

(A Pathfinder is a Pathfinder, regardless if using BFS or DFS - a strategy)

- **Factory Method**

(The user shall not be bothered with the exact object type instantiated)

Conveying less information on return values

Conveying less information on return values

```
// what do you say about this one?  
vector<int> foo() {  
    return vector {1, 2, 4};  
}
```

Conveying less information on return values

```
// is this one better?  
auto foo() {  
    return vector {1, 2, 4};  
}
```


Conveying less information on return values

```
// and what about this one?  
random_access_range_of<int> auto foo2() {  
    return vector {1, 2, 4};  
}
```

Code: <https://coliru.stacked-crooked.com/a/e07ac6d370cbbfed>

Conveying less information on return values


Using as a return value type:

- **A concept**

- => Since C++20.

- => Only for functions implemented in the header (auto must be deduced).

- => Compiler doesn't check if *usage* conforms to the concept...



Back to the hazards pointed by
Hyrum's Law.
Yes, that's a real problem.

Conveying less information on return values

Using as a return value type:

- **A concept**


- => Since C++20.

- => Only for functions implemented in the header (auto must be deduced).

- => Compiler doesn't check if *usage* conforms to the concept...

- **An interface**

- => Using dynamic virtual calls..



Back to the hazards pointed by
Hyrum's Law.

Yes, that's a real problem.

Conveying less information on return values

Using as a return value type:

- **A concept**

- => Since C++20.

- => Only for functions implemented in the header (auto must be deduced).


- => Compiler doesn't check if *usage* conforms to the concept...

- **An interface**

- => Using dynamic virtual calls.

- **A wrapper class**

- => A bit more work.



Back to the hazards pointed by
Hyrum's Law.

Yes, that's a real problem.

Hiding implementation details of return value

A note:

Beware of “leaky abstractions”

(Abstractions that expose to the user the internal behavior of the abstraction itself)

Hiding implementation details of return value

A note:

Beware of “leaky abstractions”

(Abstractions that expose to the user the internal behavior of the abstraction itself)

An example?

Hiding implementation details of return value

A note:

Beware of “leaky abstractions”

(Abstractions that expose to the user the internal behavior of the abstraction itself)

An example?

```
std::vector<bool>
```

(^ the proxy returned from)

Hiding Helper Functions

Hiding Helper Functions

```
// what do say about this one?  
class Thingy {  
public:  
    void foo();  
    // note: do not call bar before calling foo!  
    void bar();  
};
```

Hiding Helper Functions - An Attempt

```
// would this be a valid solution?  
class Thingy {  
    void foo();  
    // note: do not call bar before calling foo!  
    void bar();  
public:  
    void foobar() { foo(); bar(); }  
};
```

Hiding Helper Functions - More Generic

```
// foo and bar are related, but in a more generic way
class Thingy {
    void foo();
    // note: do not call bar before calling foo!
    void bar();
public:
    template<typename Func> void foobar(Func&& func) {
        foo();
        if(func()) bar();
    }
};
```

Code: <https://coliru.stacked-crooked.com/a/a0f86e35b362b333>

Hiding Helper Functions - Yet another approach

```
class Thingy {  
    void inner_foo() { std::cout << "foo" << std::endl; }  
    void bar() { std::cout << "bar" << std::endl; }  
public:  
    auto foo() {  
        inner_foo();  
        struct BarCallable {  
            Thingy* t;  
            void bar() { t->bar(); }  
        };  
        return BarCallable(this);  
    }  
};
```

```
int main() {  
    Thingy t;  
    t.foo().bar();  
    t.foo();  
}
```

Code: <https://coliru.stacked-crooked.com/a/c8cefb4eb38c9780>

Hiding Helper Functions

API should be hard to misuse

Using “protected”

Using “protected”

// what do say about this?

```
class Parent {  
protected:  
    Wallet wallet;  
};
```

```
class Child: public Parent {  
public:  
    void endless_celebration(long double) { wallet.draw(...); }  
};
```

Using “protected”

// A much better approach

```
class Parent {  
    Wallet wallet;  
protected:  
    short need_money(long double, const std::string& reason);  
};  
  
class Child: public Parent {  
public:  
    void endless_celebration(long double) { need_money(...); }  
};
```


Using “protected”

Strongly prefer *data members* to be **private**.

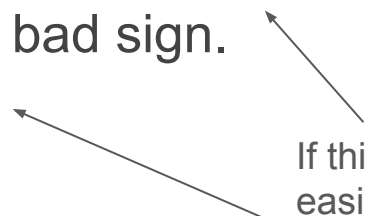
Prefer **protected *member functions*** over public, if possible.

Testing private behavior

Testing private behavior (1)

- Usually testing private behavior (e.g. the class is in state A) should be avoided.
- If there is a real need to test that, do not turn data members to be public for that purpose! Use proper member functions.
- Adding a public member function for validating class state can be useful also for logging and traceability.

Testing private behavior (2)

- A need to change the state of a class, by the test itself, not through regular flow, is usually a bad sign: we do not test actual flows (need to test error scenarios? Use mocking, create the error case via your flow).
 - Changing the code to fit your test is not a bad sign. It may improve our code!
 - Test the manipulates private data may be unstable and shaky. And usually would not test real life scenarios.
- 
- If this cannot be done easily, it may mean that your code needs redesign for better decoupling.

Testing private behavior (3)

If you still need to test private members, there are techniques for doing it, without turning them to be public.

This should not be your first choice and we will not cover it here.

(But it was covered in yesterday's talk: *Testability and API Design*, by the Bloomberg gang: John Pavan • Aram Chung • Lukas Zhao).

Use the proper context for your code

Use the proper context for your code

- Use namespaces properly to give context to your code, including hidden “detail” and unnamed (anonymous) namespaces.
- Use nested type: put types (classes, enums etc.) inside classes when relevant. Hide them in the private section if possible.
- Remember that a type declared in the private part can still be used externally, if an object of that type is provided (useful for example for inner proxies).

Use the proper context for your code

- C++20 Modules are also a tool to hide implementation details.
 - Allows for example to [hide template implementation](#) ([not much possible before C++20](#))

A talk on that right now... watch the recording:

[Why Modules? : It's not about build time](#)

Steve Downey

Last Item...

Last Item...

I owe you something.

Do you remember?

Private Token Idiom

Private Token Idiom

AKA:

- Access Token Pattern
- Passkey Pattern
- Authorization Token Pattern
- Key Token Idiom
- ...

On the naming, see:


<https://stackoverflow.com/questions/3324248/how-to-name-this-key-oriented-access-protection-pattern>

Private Token Idiom

Remember this:

```
class Foo {  
public:  
// we prefer the ctor below to be private but it doesn't compile :(  
    Foo(int value) : value(value) {}  
    static unique_ptr<Foo> create(int value) {  
        return std::make_unique<Foo>(value);  
    }  
};  
  
auto foo = Foo::create(7);
```

Can we make the ctor
private and still use
make_unique?




Code: <https://compiler-explorer.com/z/Mvn9h3G88>

Private Token Idiom

```
class Foo {  
    int value;  
    class PrivateToken {};  
public:  
    Foo(int value, PrivateToken) : value(value) {}  
    static unique_ptr<Foo> create(int value) {  
        return std::make_unique<Foo>(value, PrivateToken{});  
    }  
    int getValue() const { return value; }  
};  
  
auto foo = Foo::create(7);
```

The ctor is still in the public, but usable only with a private token.



Code: <https://compiler-explorer.com/z/M8Yof53e5>

Summary

Summary (1)

Hiding your implementation details is more than just a design recommendation for the protocol.

It will make your code **more robust** and **maintainable**.

Summary (2)

Hiding rules go well with known design principles:

Encapsulation:

Protecting Object Integrity.

Exposing necessary interfaces only.

Internal changes don't affect external code.

Easier to Debug.

Decoupling:

Reduces Dependencies: components depend only what we actually expose.

Changes in one part of the system don't necessarily affect others.

Components become more generic.

Improves Testability

Summary (3)

Having private fields is NOT enough!
(Mandatory but not sufficient).

It's context specific!

Something may be required to be public for one usage, yet, have no reason to be public for other usages.

Because:

- If they don't need it, don't give it.
- If they get it, they'll use it.
- If they use it, they'll abuse it.
- Trust no one!



Having “things” in the private is not enough!

Summary (4)

Keep Abstractions!

*Being abstract is something profoundly different from being vague.
The purpose of abstraction is not to be vague, but to create a new
semantic level in which one can be absolutely precise.*

Edsger W. Dijkstra, [EWD656](#)



Ben Deane

ABSTRACTION IS KEY

"Being abstract is something profoundly different from being vague: by abstraction - from what is uncertain or should be left open - one creates a new semantic level on which one can again be absolutely precise."

-- Edsger W Dijkstra, EWD656

84 / 101

Summary (5)

You may have reasons why to avoid hiding implementation details. 95% of the reasons are wrong.

Look for things that shouldn't be exposed.

This will improve your code and save you time and money by preventing future defects and refactoring.

Any questions before we conclude?



Bye

Core C++ 2024 - Call for Speakers is Open - Please Submit a Talk



[HOME](#) [TEAM](#)

Core C++ | 2024

November 26-28, 2024

[CALL FOR SPEAKERS](#)

[SUBMIT YOUR TALK](#)


```
class Greetings {  
    std::string greetings[] =  
        {"Thank you!"s, "תודה לכולם"s};  
public:  
    void greet() const;  
};
```