Const Correctness

Contents of this section:

FAQ What is "const correctness"?

A good thing. It means using the keyword `const` to prevent `const` objects from getting mutated.

For example, if you wanted to create a function `f()` that accepted a `std::string`, plus you want to promise callers not to change the caller's `std::string` that gets passed to `f()`, you can have `f()` receive its `std::string` parameter…

- `void f1(const std::string& s);` *// Pass by reference-to-`const`*
- `void f2(const std::string* sptr);` *// Pass by pointer-to-`const`*
- `void f3(std::string s);` *// Pass by value*

In the *pass by reference-to-`const`* and *pass by pointer-to-`const`* cases, any attempts to change the caller's `std::string` within the `f()` functions would be flagged by the compiler as an error at compile-time. This check is done entirely at compile-time: there is no run-time space or speed cost for the `const`. In the *pass by value* case (`f3()`), the called function gets a copy of the caller's `std::string`. This means that `f3()` can change its local copy, but the copy is destroyed when `f3()` returns. In particular `f3()` cannot change the caller's `std::string` object.

As an opposite example, suppose you wanted to create a function `g()` that accepted a `std::string`, but you want to let callers know that `g()` might change the caller's `std::string` object. In this case you can have `g()` receive its `std::string` parameter…

- `void g1(std::string& s);` *// Pass by reference-to-non-`const`*
- `void g2(std::string* sptr);` *// Pass by pointer-to-non-`const`*

The lack of `const` in these functions tells the compiler that they are allowed to (but are not required to) change the caller's `std::string` object. Thus they can pass their `std::string` to any of the `f()` functions, but only `f3()` (the one that receives its parameter "by value") can pass its `std::string` to `g1()` or `g2()`.

If `f1()` or `f2()` need to call either `g()` function, a local copy of the `std::string` object must be passed to the `g()` function; the parameter to `f1()` or `f2()` cannot be directly passed to either `g()` function. E.g.,

```
void g1(std::string& s);

void f1(const std::string& s)
{
  g1(s);          // Compile-time Error since s is const

  std::string localCopy = s;
  g1(localCopy);  // Okay since localCopy is not const
}
```

Naturally in the above case, any changes that `g1()` makes are made to the `localCopy` object that is local to `f1()`. In particular, no changes will be made to the `const` parameter that was passed by reference to `f1()`.

FAQ How is "`const` correctness" related to ordinary type safety?

Declaring the `const`-ness of a parameter is just another form of type safety.

If you find ordinary type safety helps you get systems correct (it does; especially in large systems), you'll find `const` correctness helps also.

The benefit of `const` correctness is that it prevents you from *inadvertently* modifying something you didn't expect would be modified. You end up needing to decorate your code with a few extra keystrokes (the `const` keyword), with the benefit that you're telling the compiler *and* other programmers some additional piece of important semantic information — information that the compiler uses to prevent mistakes and other programmers use as documentation.

Conceptually you can imagine that `const std::string`, for example, is a different class than ordinary `std::string`, since the `const` variant is conceptually missing the various mutative operations that are available in the non-`const` variant. For example, you can conceptually imagine that a `const std::string` simply doesn't have an assignment operator `+=` or any other mutative operations.

FAQ Should I try to get things `const` correct "sooner" or "later"?

At the very, very, *very* beginning.

Back-patching `const` correctness results in a snowball effect: every `const` you add "over here" requires four more to be added "over there."

Add `const` early and often.

FAQ What does "`const X* p`" mean?

It means `p` points to an object of class `X`, but `p` can't be used to change that `X` object (naturally `p` could also be `NULL`).

Read it right-to-left: "p is a pointer to an X that is constant."

For example, if class `X` has a `const` member function such as `inspect() const`, it is okay to say `p->inspect()`. But if class `X` has a non-`const` member function called `mutate()`, it is an error if you say `p->mutate()`.

Significantly, this error is caught by the compiler at compile-time — no run-time tests are done. That means `const` doesn't slow down your program and doesn't require you to write extra test-cases to check things at runtime — the compiler does the work at compile-time.

FAQ What's the difference between "`const X* p`", "`X* const p`" and "`const X* const p`"?

Read the pointer declarations right-to-left.

- `const X* p` means "`p` points to an `X` that is `const`": the `X` object can't be changed via `p`.
- `X* const p` means "`p` is a `const` pointer to an `X` that is non-`const`": you can't change the pointer `p` itself, but you can change the `X` object via `p`.
- `const X* const p` means "`p` is a `const` pointer to an `X` that is `const`": you can't change the pointer `p` itself, nor can you change the `X` object via `p`.

And, oh yea, did I mention to read your pointer declarations right-to-left?

FAQ What does "`const X& x`" mean?

It means `x` aliases an `X` object, but you can't change that `X` object via `x`.

Read it right-to-left: "`x` is a reference to an `X` that is `const`."

For example, if class `X` has a `const` member function such as `inspect() const`, it is okay to say `x.inspect()`. But if class `X` has a non-`const` member function called `mutate()`, it is an error if you say `x.mutate()`.

This is entirely symmetric with pointers to const, including the fact that the compiler does all the checking at compile-time, which means `const` doesn't slow down your program and doesn't require you to write extra test-cases to check things at runtime.

FAQ What do "`X const& x`" and "`X const* p`" mean?

`X const& x` is equivalent to `const X& x`, and `X const* x` is equivalent to `const X* x`.

Some people prefer the `const`-on-the-right style, calling it "consistent `const`" or, using a term coined by Simon Brand, "East `const`." Indeed the "East `const`" style can be more consistent than the alternative: the "East `const`" style *always* puts the `const` on the right of what it constifies, whereas the other style *sometimes* puts the `const` on the left and *sometimes* on the right (for `const` pointer declarations and `const` member functions).

With the "East `const`" style, a local variable that is `const` is defined with the `const` on the right: `int const a = 42;`. Similarly a `static` variable that is `const` is defined as `static double const x = 3.14;`. Basically every `const` ends up on the right of the thing it constifies, including the `const` that is *required* to be on the right: `const` pointer declarations and with a `const` member function.

The "East `const`" style is also less confusing when used with type aliases: Why do `foo` and `bar` have different types here?

```
using X_ptr = X*;

const X_ptr foo;
const X* bar;
```

Using the "East `const`" style makes this clearer:

```
using X_ptr = X*;

X_ptr const foo;
X* const foobar;
X const* bar;
```

It is clearer here that `foo` and `foobar` are the same type and that `bar` is a different type.

The "East `const`" style is also more consistent with pointer declarations. Contrast the traditional style:

```
const X** foo;
const X* const* bar;
const X* const* const baz;
```

with the "East `const`" style

```
X const** foo;
X const* const* bar;
X const* const* const baz;
```

Despite these benefits, the `const`-on-the-right style is not yet popular, so legacy code tends to have the traditional style.

FAQ Does "`X& const x`" make any sense?

No, it is nonsense.

To find out what the above declaration means, read it right-to-left: "`x` is a `const` reference to a `X`". But that is redundant — references are always `const`, in the sense that you can never reseat a reference to make it refer to a different object. Never. With or without the `const`.

In other words, "`X& const x`" is functionally equivalent to "`X& x`". Since you're gaining nothing by adding the `const` after the `&`, you shouldn't add it: it will confuse people — the `const` will make some people think that the `X` is `const`, as if you had said "`const X& x`".

FAQ What is a "`const` member function"?

A member function that inspects (rather than mutates) its object.

A `const` member function is indicated by a `const` suffix just after the member function's parameter list. Member functions with a `const` suffix are called "`const` member functions" or "inspectors." Member functions without a `const` suffix are called "non-`const` member functions" or "mutators."

```
class Fred {
public:
  void inspect() const;   // This member promises NOT to change *this
  void mutate();          // This member function might change *this
};

void userCode(Fred& changeable, const Fred& unchangeable)
{
  changeable.inspect();   // Okay: doesn't change a changeable object
  changeable.mutate();    // Okay: changes a changeable object

  unchangeable.inspect(); // Okay: doesn't change an unchangeable obj
  unchangeable.mutate();  // ERROR: attempt to change unchangeable ob
}
```

The attempt to call `unchangeable.mutate()` is an error caught at compile time. There is no runtime space or speed penalty for `const`, and you don't need to write test-cases to check it at runtime.

The trailing `const` on `inspect()` member function should be used to mean the method won't change the object's *abstract* (client-visible) state. That is slightly different from saying the method won't change the "raw bits" of the object's `struct`. C++ compilers aren't allowed to take the "bitwise" interpretation unless they can solve the aliasing problem, which normally can't be solved (i.e., a non-`const` alias could exist which could modify the state of the object). Another (important) insight from this aliasing issue: pointing at an object with a pointer-to-`const` doesn't guarantee that the object won't change; it merely promises that the object won't change *via that pointer*.

FAQ What is the relationship between a return-by-reference and a `const` member function?

If you want to return a member of your `this` object by reference from an inspector method, you should return it using reference-to-

const (`const X& inspect() const`) or by value (`X inspect() const`).

```
class Person {
public:
  const std::string& name_good() const;  // Right: the caller can't c
  std::string& name_evil() const;         // Wrong: the caller can cha
  int age() const;                        // Also right: the caller ca
  // ...
};

void myCode(const Person& p)  // myCode() promises not to change the
{
  p.name_evil() = "Igor";     // But myCode() changed it anyway!!
}
```

The good news is that the compiler will *often* catch you if you get this wrong. In particular, if you accidentally return a member of your `this` object by non-`const` reference, such as in `Person::name_evil()` above, the compiler will *often* detect it and give you a compile-time error while compiling the innards of, in this case, `Person::name_evil()`.

The bad news is that the compiler *won't always* catch you: there are some cases where the compiler simply won't ever give you a compile-time error message.

Translation: you need to *think*. If that scares you, find another line of work; "think" is not a four-letter word.

Remember the "`const` philosophy" spread throughout this section: a `const` member function must not change (or allow a caller to change) the `this` object's *logical* state (AKA *abstract* state AKA *meaningwise* state). Think of what an object *means*, not how it is internally implemented. A Person's age and name are logically part of the Person, but the Person's neighbor and employer are not. An inspector method that returns part of the `this` object's logical / abstract / meaningwise state *must not* return a non-`const` pointer (or reference) to that part, independent of whether that part is internally implemented as a direct data-member physically embedded within the `this` object or some other way.

FAQ What's the deal with "`const`-overloading"?

`const` overloading helps you achieve `const` correctness.

`const` overloading is when you have an inspector method and a mutator method with the same name and the same number of and types of parameters. The two distinct methods differ only in that the inspector is `const` and the mutator is non-`const`.

The most common use of `const` overloading is with the subscript operator. You should generally try to use one of the standard container templates, such as `std::vector`, but if you need to create your own class that has a subscript operator, here's the rule of thumb: **subscript operators often come in pairs.**

```
class Fred { /*...*/ };

class MyFredList {
public:
  const Fred& operator[] (unsigned index) const;  // Subscript operat
  Fred&       operator[] (unsigned index);        // Subscript operat
  // ...
};
```

The `const` subscript operator returns a `const`-reference, so the compiler will prevent callers from inadvertently mutating/changing the `Fred`. The non-`const` subscript operator returns a non-`const` reference, which is your way of telling your callers (and the compiler) that your callers are allowed to modify the `Fred` object.

When a user of your `MyFredList` class calls the subscript operator, the compiler selects which overload to call based on the constness of *their* `MyFredList`. If the caller has a `MyFredList a` or `MyFredList& a`, then `a[3]` will call the non-`const` subscript operator, and the caller will end up with a non-`const` reference to a `Fred`:

For example, suppose `class Fred` has an inspector-method `inspect() const` and a mutator-method `mutate()`:

```
void f(MyFredList& a)  // The MyFredList is non-const
{
  // Okay to call methods that inspect (Look but not mutate/change) t
  Fred x = a[3];       // Doesn't change to the Fred at a[3]: merely
  a[3].inspect();      // Doesn't change to the Fred at a[3]: inspect

  // Okay to call methods that DO change the Fred at a[3]:
  Fred y;
  a[3] = y;            // Changes the Fred at a[3]
  a[3].mutate();       // Changes the Fred at a[3]: mutate() is a mut
}
```

However if the caller has a `const MyFredList a` or `const MyFredList& a`, then `a[3]` will call the `const` subscript operator, and the caller will end up with a `const` reference to a `Fred`. This allows the caller to inspect the `Fred` at `a[3]`, but it prevents the caller from inadvertently mutating/changing the `Fred` at `a[3]`.

```
void f(const MyFredList& a)  // The MyFredList is const
{
  // Okay to call methods that DON'T change the Fred at a[3]:
  Fred x = a[3];
  a[3].inspect();

  // Compile-time error (fortunately!) if you try to mutate/change th
  Fred y;
  a[3] = y;       // Fortunately(!) the compiler catches this error a
  a[3].mutate();  // Fortunately(!) the compiler catches this error a
}
```

Const overloading for subscript- and funcall-operators is illustrated here, here, here, here, and here.

You can, of course, also use `const`-overloading for things other than the subscript operator.

FAQ How can it help me design better classes if I distinguish *logical state* from *physical state*?

Because that encourages you to design your classes from the outside-in rather than from the inside-out, which in turn makes your classes and objects easier to understand and use, more intuitive, less error prone, and faster. (Okay, that's a slight over-simplification. To understand all the if's and's and but's, you'll just have to read the rest of this answer!)

Let's understand this from the inside-out — you will (should) design your classes from the outside-in, but if you're new to this concept, it's easier to understand from the inside-out.

On the inside, your objects have physical (or concrete or bitwise) state. This is the state that's easy for programmers to see and understand; it's the state that would be there if the class were just a C-style `struct`.

On the outside, your objects have users of your class, and these users are restricted to using only `public` member functions and `friends`. These external users also perceive the object as having state, for example, if the object is of class `Rectangle` with methods `width()`, `height()` and `area()`, your users would say that those three are all part of the object's logical (or abstract or meaningwise) state. To an external user, the `Rectangle` object actually has an area, even if that area is computed on the fly (e.g., if the `area()` method returns the product of the object's width and height). In fact, and this is the important point, your users don't know and don't care how you implement any of these methods; your users still perceive, from their perspective, that your object logically has a meaningwise state of width, height, and area.

The `area()` example shows a case where the logical state can contain elements that are not directly realized in the physical state. The opposite is also true: classes sometimes intentionally hide part of their objects' physical (concrete, bitwise) state from users — they intentionally do not provide any `public` member functions or `friends` that would allow users to read or write or even know about this hidden state. That means there are bits in the object's physical state that have no corresponding elements in the object's logical state.

As an example of this latter case, a collection-object might cache its last lookup in hopes of improving the performance of its next lookup. This cache is certainly part of the object's physical state, but there it is an internal implementation detail that will probably not be exposed to users — it will probably not be part of the object's logical state. Telling what's what is easy if you think from the outside-in: if the collection-object's users have no way to check the state of the cache itself, then the cache is *transparent*, and is not part of the object's logical state.

FAQ Should the `constness` of my `public` member functions be based on what the method does to the object's *logical state*, or *physical state*?

Logical.

There's no way to make this next part easy. It *is* going to hurt. Best recommendation is to sit down. And please, for your safety, make sure there are no sharp implements nearby.

Let's go back to the collection-object example. Remember: there's a lookup method that caches the last lookup in hopes to speed up future lookups.

Let's state what is probably obvious: assume that the lookup method makes *no* changes to *any* of the collection-object's logical state.

So… the time has come to hurt you. Are you ready?

Here comes: if the lookup method does not make any change to any of the collection-object's logical state, but it *does* change the collection-object's *physical* state (it makes a very real change to the very real cache), should the lookup method be `const`?

The answer is a resounding Yes. (There are exceptions to every rule, so "Yes" should really have an asterisk next to it, but the vast majority of the time, the answer is Yes.)

This is all about "*logical* `const`" over "*physical* `const`." It means the decision about whether to decorate a method with `const` should hinge primarily on whether that method leaves the *logical* state unchanged, *irrespective* (are you sitting down?) (you might want to sit down) *irrespective* of whether the method happens to make very real changes to the object's very real physical state.

In case that didn't sink in, or in case you are not yet in pain, let's tease it apart into two cases:

- If a method changes any part of the object's logical state, it logically is a mutator; it should not be `const` *even if* (as actually happens!) the method doesn't change any physical bits of the object's concrete state.
- Conversely, a method is logically an inspector and should be `const` if it never changes any part of the object's logical state, even if (as actually happens!) the method changes physical bits of the object's concrete state.

If you're confused, read it again.

If you're not confused but are angry, good: you may not like it yet, but at least you understand it. Take a deep breath and repeat after me: *"The `const`ness of a method should makes sense from outside the object."*

If you're still angry, repeat this three times: *"The constness of a method must make sense to the object's users, and those users can see only the object's logical state."*

If you're still angry, sorry, it is what it is. Suck it up and live with it. Yes, there will be exceptions; every rule has them. But as a rule, in the main, this *logical* `const` notion is good for you and good for your software.

One more thing. This is going to get inane, but let's be precise about whether a method changes the object's logical state. If you are *outside* the class — you are a normal user, every experiment you could perform (every method or sequence of methods you call) would have the *same results* (same return values, same exceptions or lack of exceptions) irrespective of whether you first called that lookup method. If the lookup function changed *any* future behavior of *any* future method (not just making it faster but changed the outcome, changed the return value, changed the exception), then the lookup method changed the object's logical state — it is a mutuator. But if the lookup method changed nothing other than perhaps making some things faster, then it is an inspector.

FAQ What do I do if I want a `const` member function to make an "invisible" change to a data member?

Use `mutable` (or, as a last resort, use `const_cast`).

A small percentage of inspectors need to make changes to an object's physical state that cannot be observed by external users — changes to the physical but not logical state.

For example, the collection-object discussed earlier cached its last lookup in hopes of improving the performance of its next lookup. Since the cache, in this example, cannot be directly observed by any part of the collection-object's public interface (other than timing), its

existence and state is not part of the object's logical state, so changes to it are invisible to external users. The lookup method is an inspector since it never changes the object's logical state, irrespective of the fact that, at least for the present implementation, it changes the object's physical state.

When methods change the physical but not logical state, the method should generally be marked as `const` since it really is an inspector-method. That creates a problem: when the compiler sees your `const` method changing the physical state of the `this` object, it will complain — it will give your code an error message.

The C++ compiler language uses the `mutable` keyword to help you embrace this *logical* `const` notion. In this case, you would mark the cache with the `mutable` keyword, that way the compiler knows it is allowed to change inside a `const` method or via any other `const` pointer or reference. In our lingo, the `mutable` keyword marks those portions of the object's physical state which are not part of the logical state.

The `mutable` keyword goes just before the data member's declaration, that is, the same place where you could put `const`. The other approach, not preferred, is to cast away the `const`'ness of the `this` pointer, probably via the `const_cast` keyword:

```
Set* self = const_cast<Set*>(this);
  // See the NOTE below before doing this!
```

After this line, `self` will have the same bits as `this`, that is, `self == this`, but `self` is a `Set*` rather than a `const Set*` (technically `this` is a `const Set* const`, but the right-most `const` is irrelevant to this discussion). That means you can use `self` to modify the object pointed to by `this`.

NOTE: there is an extremely unlikely error that can occur with `const_cast`. It only happens when three very rare things are combined at the same time: a data member that ought to be `mutable` (such as is discussed above), a compiler that doesn't support the `mutable` keyword and/or a programmer who doesn't use it, and an object that was originally defined to be `const` (as opposed to a normal, non-`const` object that is pointed to by a

pointer-to-`const`). Although this combination is so rare that it may never happen to you, if it ever did happen, the code may not work (the Standard says the behavior is undefined).

If you ever want to use `const_cast`, use `mutable` instead. In other words, if you ever need to change a member of an object, and that object is pointed to by a pointer-to-`const`, the safest and simplest thing to do is add `mutable` to the member's declaration. You can use `const_cast` if you are *sure* that the actual object isn't `const` (e.g., if you are sure the object is declared something like this: `Set s;`), but if the object itself might be `const` (e.g., if it might be declared like: `const Set s;`), use `mutable` rather than `const_cast`.

Please don't write saying version *X* of compiler *Y* on machine *Z* lets you change a non-`mutable` member of a `const` object. I don't care — it is illegal according to the language and your code will probably fail on a different compiler or even a different version (an upgrade) of the same compiler. Just say no. Use `mutable` instead. Write code that is *guaranteed to work,* not code that *doesn't seem to break.*

FAQ Does `const_cast` mean lost optimization opportunities?

In theory, yes; in practice, no.

Even if the language outlawed `const_cast`, the only way to avoid flushing the register cache across a `const` member function call would be to solve the aliasing problem (i.e., to prove that there are no non-`const` pointers that point to the object). This can happen only in rare cases (when the object is constructed in the scope of the `const` member function invocation, and when all the non-`const` member function invocations between the object's construction and the `const` member function invocation are statically bound, and when every one of these invocations is also `inline`d, and when the constructor itself is `inline`d, and when any member functions the constructor calls are `inline`).

FAQ Why does the compiler allow me to change an `int` after I've pointed at it with a `const`

`int*`?

Because "`const int* p`" means "`p` promises not to change the `*p`," *not* "`*p` promises not to change."

Causing a `const int*` to point to an `int` doesn't `const`-ify the `int`. The `int` can't be changed via the `const int*`, but if someone else has an `int*` (note: no `const`) that points to ("aliases") the same `int`, then that `int*` can be used to change the `int`. For example:

```
void f(const int* p1, int* p2)
{
  int i = *p1;          // Get the (original) value of *p1
  *p2 = 7;              // If p1 == p2, this will also change *p1
  int j = *p1;          // Get the (possibly new) value of *p1
  if (i != j) {
    std::cout << "*p1 changed, but it didn't change via pointer p1!\n
    assert(p1 == p2);  // This is the only way *p1 could be different
  }
}

int main()
{
  int x = 5;
  f(&x, &x);            // This is perfectly legal (and even moral!)
  // ...
}
```

Note that `main()` and `f(const int*,int*)` could be in different compilation units that are compiled on different days of the week. In that case there is no way the compiler can possibly detect the aliasing at compile time. Therefore there is no way we could make a language rule that prohibits this sort of thing. In fact, we wouldn't even want to make such a rule, since in general it's considered a feature that you can have many pointers pointing to the same thing. The fact that one of those pointers promises not to change the underlying "thing" is just a promise made by the *pointer;* it's *not* a promise made by the "thing".

FAQ Does "`const Fred* p`" mean that `*p` can't change?

No! (This is related to the FAQ about aliasing of `int` pointers.)

"`const Fred* p`" means that the `Fred` can't be changed via pointer `p`, but there might be other ways to get at the object without going through a `const` (such as an aliased non-`const` pointer such

as a `Fred*`). For example, if you have two pointers "`const Fred* p`" and "`Fred* q`" that point to the same `Fred` object (aliasing), pointer `q` can be used to change the `Fred` object but pointer `p` cannot.

```
class Fred {
public:
  void inspect() const;   // A const member function
  void mutate();          // A non-const member function
};

int main()
{
  Fred f;
  const Fred* p = &f;
  Fred*       q = &f;

  p->inspect();    // Okay: No change to *p
  p->mutate();     // Error: Can't change *p via p

  q->inspect();    // Okay: q is allowed to inspect the object
  q->mutate();     // Okay: q is allowed to mutate the object

  f.inspect();     // Okay: f is allowed to inspect the object
  f.mutate();      // Okay: f is allowed to mutate the object

  // ...
}
```

FAQ Why am I getting an error converting a `Foo**` → `const Foo**`?

Because converting `Foo**` → `const Foo**` would be invalid and dangerous.

C++ allows the (safe) conversion `Foo*` → `Foo const*`, but gives an error if you try to implicitly convert `Foo**` → `const Foo**`.

The rationale for why that error is a good thing is given below. But first, here is the most common solution: simply change `const Foo**` to `const Foo* const*`:

```
class Foo { /* ... */ };

void f(const Foo** p);
void g(const Foo* const* p);

int main()
{
  Foo** p = /*...*/;
  // ...
  f(p);  // ERROR: it's illegal and immoral to convert Foo** to const
  g(p);  // Okay: it's legal and moral to convert Foo** to const Foo*
  // ...
}
```

The reason the conversion from `Foo**` → `const Foo**` is dangerous is that it would let you silently and accidentally modify a `const Foo` object without a cast:

```
class Foo {
public:
  void modify();  // make some modification to the this object
};

int main()
{
  const Foo x;
  Foo* p;
  const Foo** q = &p;  // q now points to p; this is (fortunately!) a
  *q = &x;             // p now points to x
  p->modify();         // Ouch: modifies a const Foo!!
  // ...
}
```

If the `q = &p` line were legal, `q` would be pointing at `p`. The next line, `*q = &x`, changes `p` itself (since `*q` is `p`) to point at `x`. That would be a bad thing, since we would have lost the `const` qualifier: `p` is a `Foo*` but `x` is a `const Foo`. The `p->modify()` line exploits `p`'s ability to modify its referent, which is the real problem, since we ended up modifying a `const Foo`.

By way of analogy, if you hide a criminal under a lawful disguise, he can then exploit the trust given to that disguise. That's bad.

Thankfully C++ prevents you from doing this: the line `q = &p` is flagged by the C++ compiler as a compile-time error. Reminder: *please* do *not* pointer-cast your way around that compile-time error message. Just Say No!

(Note: there is a conceptual similarity between this and the prohibition against converting `Derived**` to `Base**`.)