const-correctness in C++

Matthias Kretz

Frankfurt Institute for Advanced Studies
Institute for Computer Science
Goethe University Frankfurt

2015-04-01

HIC for FAIR Helmholtz International Center

FIAS Frankfurt Institute for Advanced Studies

HGS-HIRe for FAIR Helmholtz Graduate School for Hadron and Ion Research

Acknowledgements

My sources for inspiration:

- https://isocpp.org/wiki/faq/const-correctness
- https://en.wikipedia.org/wiki/Const-correctness
- http://www.cprogramming.com/tutorial/const_correctness.html

Where can you use `const`?

Where can you use `const`?

- variables (global, local, member)
- function parameters
- member functions
- type aliases (`typedef`, `using`)

It's part of the type!

```
is_same<int, const int>::value == false
is_same<int, const int&>::value == false
is_same<int*, const int*>::value == false
is_same<const int*, int const*>::value == true
is_same<int*, int* const>::value == false
is_same<int, int&>::value == false
is_same<int, volatile int>::value == false
is_same<const int, volatile int>::value == false
is_same<const int, const volatile int>::value == false
```

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
- `f(1);`
- `int n = 1; f(n);`

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
  OK: takes the address of `n` to pass a reference to `const int&`.
- `f(1);`
- `int n = 1; f(n);`

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
  OK: takes the address of `n` to pass a reference to `const int&`.
- `f(1);`

- `int n = 1; f(n);`

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
  OK: takes the address of `n` to pass a reference to `const int&`.
- `f(1);`
  OK: Compiler generates an anonymous constant 1 in memory.
- `int n = 1; f(n);`

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
  OK: takes the address of `n` to pass a reference to `const int&`.
- `f(1);`
  OK: Compiler generates an anonymous constant 1 in memory.
- `int n = 1; f(n);`

Function Arguments allow for Conversions

Given the function `f(const int &)`, are the following calls valid?

- `const int n = 1; f(n);`
  OK: takes the address of `n` to pass a reference to `const int&`.
- `f(1);`
  OK: Compiler generates an anonymous constant 1 in memory.
- `int n = 1; f(n);`
  OK: `int&` is implicitly converted to `const int&`. A const-ref to a non-const variable is always fine.

Tell me what you see:

- `const int a;`
- `const int b = 1;`
- `int c = b;`
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
- `int c = b;`
- `int &d = b;`
- `const int &e = c;`

"If a program calls for the default initialization of an object of a const-qualified type `T`, `T` shall be a class type with a user-provided default constructor."

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
- `int c = b;`
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
  copy value to a non-const variable
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
  copy value to a non-const variable
- `int &d = b;`
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
  copy value to a non-const variable
- `int &d = b;`
  error: non-const reference to immutable variable
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
  copy value to a non-const variable
- `int &d = b;`
  error: non-const reference to immutable variable
- `const int &e = c;`

Tell me what you see:

- `const int a;`
  error: missing initialization
- `const int b = 1;`
  normal constant, consider `constexpr int b = 1;` instead
- `int c = b;`
  copy value to a non-const variable
- `int &d = b;`
  error: non-const reference to immutable variable
- `const int &e = c;`
  immutable reference to `c` (which may still be modified)

Digression: `constexpr`

- `constexpr` is a new keyword since C++11
- short for: *constant expression*
- use it for constants that can be evaluated at compile time
- template arguments must be constant expressions
- no storage & linkage requirements unless the address of a `constexpr` "variable" is taken

const Member Functions

```
struct A {
  void f();       // (1)
  void f() const; // (2)
};
A a;
const A c;
```

- `a.f()` calls
- `c.f()` calls

`const` Member Functions

```
struct A {
  void f();       // (1)
  void f() const; // (2)
};
A a;
const A c;
```

- `a.f()` calls `(1)`
- `c.f()` calls `(2)`

The function overloads match on the `this` pointer. Consider that the compiler actually emits the functions `void A::f(A *this)` and `void A::f(const A *this)` for `A::f`.

So what does it do?

...besides modifying the type

- `const` builtin types cannot be assigned to
- non-const implicitly converts to `const`
- `const` cannot implicitly convert to non-const
- better: only `const_cast` can cast away `const`
  get rid of C-casts! (`-Wold-style-cast`)
- Note, you can cast away `const`!
- You can write
  `struct X { void operator=(T) const; };` .
  And thus have a **`const X`** variable that is assignable.
- As so often, you can use good to create bad ...

Why `const` if there's no guarantee that it stays `const`?

So what does it do?

...besides modifying the type

- `const` builtin types cannot be assigned to
- non-const implicitly converts to `const`
- `const` cannot implicitly convert to non-const
- better: only `const_cast` can cast away `const`
  get rid of C-casts! (`-Wold-style-cast`)
- Note, you can cast away `const`!
- You can write
  `struct X { void operator=(T) const; };` .
  And thus have a **`const X`** variable that is assignable.
- As so often, you can use good to create bad ...

Why `const` if there's no guarantee that it stays `const`?

`mutable`

We need to cover one more.

mutable

What does `mutable` do?

`mutable`

What does `mutable` do?

`mutable` makes member variables mutable in `const` member functions.

What did the C++ designers intend when they conceived `const`?

`const` means logically constant, not physically constant.

physically constant: the bits in memory/registers do not change

logically constant: the observable state of an object/variable does not change

The class interface designer is responsible for correctly implementing *logically constant* semantics.

An Example

```cpp
class A {
  double x = 1.;
public:
  double value() const { return x; }
  void setValue(double xx) { x = xx; }
  double transformed() const { return expensiveFunction(x); }
};
```

- This interface is *const-correct*:
  - `A::value` and `A::transformed` keep the state constant
  - `A::setValue` modifies the state
- Consider a typical use pattern of zero or many calls to `A::transformed`

  zero better never evaluate `expensiveFunction`

  many better evaluate `expensiveFunction` only once per new `x`

An Example cont.

```
class A2 {
  double x = 1.;
  static constexpr double dirty_value =
      std::numeric_limits<double>::infinity();
  double cached = dirty_value;
public:
  double value() const { return x; }
  void setValue(double xx) {
    x = xx;
    cached = dirty_value;
  }
  double transformed() /* not const! */ {
    if (cached == dirty_value) {
      cached = expensiveFunction(x);
    }
    return cached;
  }
};
```

An Example cont..

- The interface of **`A2`** is not const-correct!
- `A2::transformed` does not change the observable state
  $\Rightarrow$ it should be `const`.
- `A2::transformed` requires callers to to use a non-const object.
  $\Rightarrow$ removal of `const` from other logically constant functions (Which might even appear physically constant in their implementation)

Solutions?

Solutions

- `const_cast`
- `mutable`

Always prefer `mutable` over `const_cast`!

An Example cont...

```
class A3 {
  double x = 1.;
  static constexpr double dirty_value =
      std::numeric_limits<double>::infinity();
  mutable double cached = dirty_value;
public:
  double value() const { return x; }
  void setValue(double xx) {
    x = xx;
    cached = dirty_value;
  }
  double transformed() const {       // keeps logical state
    if (cached == dirty_value) {
      cached = expensiveFunction(x); // modifies physical state
    }
    return cached;
  }
};
```

What does the interface of `A3` tell you?

`const` implicitly documents the interface

- That `A3::transformed` is `const` says:
  - repeated calls to `A3::transformed` return the same value
- That `A3::value` is `const` says:
  - interleaving calls to `A3::value` does not change `A3::transformed`
- That `A3::setValue` is *not* `const` says:
  - after the call the state of the object has changed
  - return values of member functions may change as a result

`const` implicitly documents the interface

- That `A3::transformed` is `const` says:
  - repeated calls to `A3::transformed` return the same value
- That `A3::value` is `const` says:
  - interleaving calls to `A3::value` does not change `A3::transformed`
- That `A3::setValue` is *not* `const` says:
  - after the call the state of the object has changed
  - return values of member functions may change as a result

However, the compiler cannot rely on this for optimization.

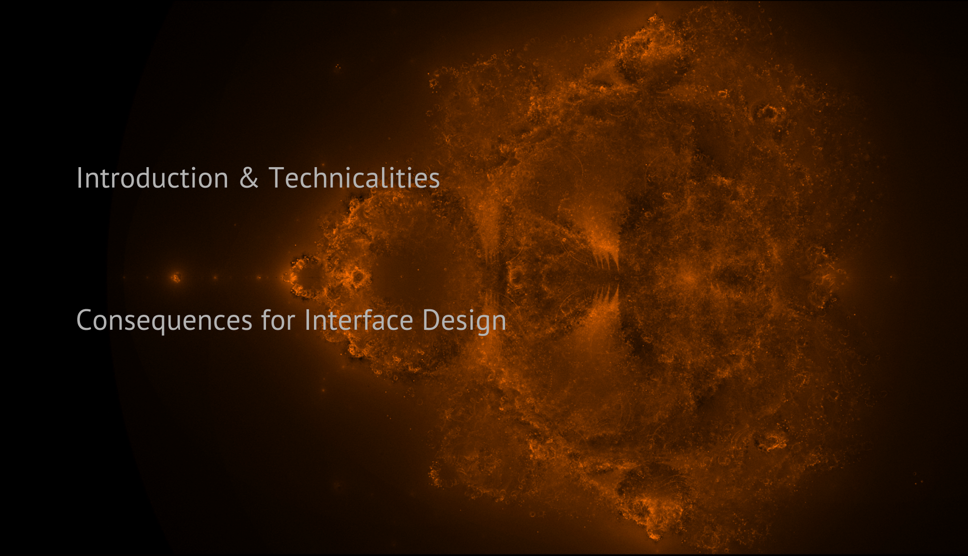
Consider global variables, `mutable`, and `const_cast` ...

Takeaways

1. `const` means *logically constant*.
2. Decide on `const`ness of member functions based on *logical state*.
3. Use `const` to document interfaces.
4. Use `const` to make your interfaces harder/impossible to use incorrectly.
5. Design const-correct code from the beginning of the project.
6. Use `constexpr` for constants that can be evaluated at compile time.

A different talk should add:

7. `const` member functions need to be thread-safe.
8. `mutable` member variable access needs to be atomic.



Introduction & Technicalities

Consequences for Interface Design