accu 2022

C++ CONST CORRECTNESS REFRESHER

VLADIMIR VISHNEVSKII

Const-correctness

- Correct assignment of constness property (using “const” keyword)
- Consistent management of constness for owned/shared objects in complex types

Why is it important ?

- Constness can enforce immutability (enforces for fundamental types and pointers)

  Immutability gives the following advantages:

  - no unintentional/silent modification (reliability, safety)
  - no race condition (reliability, safety)
  - clarifies purpose for reader/reviewer (maintainability, review quality)
  - clarifies purpose for compiler to aid optimization (performance)

- Extras for complex types:
  - allows to separate interface methods into mutating and read-only
  - allows (**and requires**) to implement various encapsulation strategies

Guidelines and standards

C++ Core guidelines:
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const

Summary:

- Con.1: By default, make objects immutable
- Con.2: By default, make member functions `const`
- Con.3: By default, pass pointers and references to `const`s
- Con.4: Use `const` to define objects with values that do not change after construction
- Con.5: Use `constexpr` for values that can be computed at compile time

Guidelines and standards

AUTOSAR C++14:
https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf

Rule A7-1-1 (required, implementation, automated)
Constexpr or const specifiers shall be used for immutable data declaration.

Rule A7-1-2 (required, implementation, automated)
The constexpr specifier shall be used for values that can be determined at compile time.

Agenda

- What **can** and **should** be specified as **const**?
- How constness **can** and **should** be managed in complex types?

Variables

- Variable can be modified:

```
int i = calculate_value();
i = 10;
i++;
```

Variables

- Can be modified accidentally:

```
int i = calculate_value();
if (i = 10) {}
//...
if (auto i = calculate_value(); i = 10) {}
```

Valid syntax

Warning in ***gcc*** with ***-Wall***

Warning in ***clang*** by default

Variables

- Can be modified accidentally:

```
int i = calculate_value();
if (i = 10) {}
//...

if (auto i = calculate_value(); i = 10) {}
```

auto is "responsible" for type only

Variables

- Can be modified accidentally:

```
int i = calculate_value();

if (i = 10) {}
//...


if (auto i = calculate_value(); i = 10) {}
```

Same diagnostics

Variables

- The variable can be modified in the function:

```
size_t size = calculate_size();
auto b = allocate_buffer_of_size(size);
```

Variables

- The variable can be modified in the function:

```
size_t size = calculate_size();
auto b = allocate_buffer_of_size(size);
```

> No guarantee that the value of ***size*** was not modified

Variables

- The variable can be modified in the function:

```
size_t size = calculate_size();
auto b = allocate_buffer_of_size(size);

void *allocate_buffer_of_size(size_t &adjusted_size);
```

Input/output parameter. Poor design, but valid syntax

Variables

- ~~Can be modified accidentally~~:

```
const int i = calculate_value();

if (i = 10) {}
//...

if (const auto i = calculate_value(); i = 10) {}
```

Declares *i* to be immutable

Variables

- ~~The variable can be modified in the function:~~

```
const size_t size = calculate_size();
auto b = allocate_buffer_of_size(size);

void *allocate_buffer_of_size(int &adjusted_size);
```

Will not bind to non-const reference

Variables

- constexpr for values that can be evaluated at compile time:

```
constexpr int i = 10 * 10;


constexpr auto j = 10 * 10;
```

Pointers

- Can be accidentally modified:

```
const int *ptr = ...;
if (ptr = nullptr) {
    //...
}

int *counter_ptr = ...;
while (*counter_ptr--) {
    //...
}
```

Assignment instead of comparison. Regardless of const qualification of the referenced type

Pointers

- Can be accidentally modified:

```
const int *ptr = ...;
if (ptr = nullptr) {
    //...
}

int *counter_ptr = ...;
while (*counter_ptr--) {
    //...
}
```

Decrement is performed first

Pointers

- ~~Can be accidentally modified:~~

```
const int *const ptr = ...;
if (ptr = nullptr) {
    //...
}

int *const counter_ptr = ...;
while (*counter_ptr--) {
    //...
}
```

Immutable pointer to mutable data

Auto type deduction

- ***auto*** deduction rules should be considered:

```
const int &get_value();
auto v = get_value();
if (v = 10) {}
```

auto deduction drops reference and const, so the **mutable copy** will be created

Auto type deduction

- ***auto*** deduction rules should be considered:

```
const int &get_value();
const auto v = get_value();
if (v = 10) {}
```

> If immutable copy is required.
> Or explicit ***<const> auto &***
> if reference is needed.

Auto type deduction

- Pointers and auto:

```
int obj{};
auto ptr = &obj;
//...
const auto ptr = &obj;
const auto *ptr = &obj;
const auto *const ptr = &obj;
```

```
Implies int *ptr = &obj;
Non-const pointer to non-const
object
```

Auto type deduction

- Pointers and auto:

```
int obj{};
auto ptr = &obj;
//...
const auto ptr = &obj;
const auto *ptr = &obj;
const auto *const ptr = &obj;
```

Implies `int *const ptr = &obj;`

Auto type deduction

- Pointers and auto:

```
int obj{};
auto ptr = &obj;
//...
const auto ptr = &obj;
const auto *ptr = &obj;
const auto *const ptr = &obj;
```

Implies `const int *ptr = &obj;`

Auto type deduction

- Pointers and auto:

```
int obj{};
auto ptr = &obj;
//...
const auto ptr = &obj;
const auto *ptr = &obj;
const auto *const ptr = &obj;
```

Implies `const int *const ptr = &obj;`

STL Iterators

- *iterator* references mutable object;
- *const_iterator* references immutable object;

- Objects of both types can be unintentionally modified:

```
const std::vector<int> v{1, 2, 3};
auto begin = v.begin();
auto end = v.end();

some_custom_algorithm(begin, end);
```

> No guarantee that iterator itself (***std::vector::const_iterator*** in this case) will not changed in the function

STL Iterators

- ~~Objects of both types can be unintentionally modified:~~

```
template <typename It>
void some_custom_algorithm(It &begin, const It &end) {
    while (begin != end) ++begin;
}

const std::vector<int> v{1, 2, 3};
const auto begin = v.begin();
const auto end = v.end();

some_custom_algorithm(begin, end);
```

Will not compile

Smart pointers

- Can be accidentally reassigned:

```
auto ptr = std::make_shared<int>(10);
if (ptr = nullptr) {
    // ...
}
```

Smart pointers

- Can be accidentally reassigned:

```
auto ptr = std::make_shared<int>(10);
if (ptr = nullptr) {
    // ...
}
```

Similar to plain pointers

Smart pointers

- ~~Can be accidentally reassigned:~~

```
const auto ptr = std::make_shared<int>(10);
if (ptr = nullptr) {
    // ...
}
```

Conts shared_ptr allows modification of referenced object.

- Const ***shared_ptr*** allows modification of the referenced data:

std::shared_ptr<T>::operator*, std::shared_ptr<T>::operator->

`T& operator*() const noexcept;`	(1)	(since C++11)
`T* operator->() const noexcept;`	(2)	(since C++11)

Function parameters

- Similar to variables:

```
void increment_n_times(int n, int *obj) {
    if (n = 0) return;
    for (auto i = 0; i < n; ++i) *obj++;
}
```

Function parameters

- Similar to variables:

```
void increment_n_times(int n, int *obj) {
    if (n = 0) return;
    for (auto i = 0; i < n; ++i) *obj++;
}
```

Issues from previous examples

Function parameters

- Solution is to make objects that should not be modified const in function/method definition:

```
void increment_n_times(int n, int *obj);
// ...
void increment_n_times(const int n, int *const obj) {
    if (n = 0) return;
    for (auto i = 0; i < n; ++i) *obj++;
}
```

depending on how the parameter will be used in definition

Function parameters

- Solution is to make objects that should not be modified const in function/method definition:

```
void increment_n_times(int n, int *obj);
// ...
void increment_n_times(const int n, int *const obj) {
    if (n = 0) return;
    for (auto i = 0; i < n; ++i) *obj++;
}
```

cv-qualification is ignored between declaration and definition

Initialization

- Conditional initialization:

```
int i = 1;
if (should_be_two_not_one)
    i = 2;
```

Initialization

- Conditional initialization:

```
int i = 1;
if (should_be_two_not_one)
    i = 2;

const int i = should_be_two_not_one ? 2 : 1
```

operator :? can be used

Initialization

- Immediately invoked lambda for more complex conditions:

```
const int v = get_some_value();
const auto n_v = [v] { // or [&v] for complex types
    if (v < 100)
        return 0;
    else if (v >= 100 && v < 1000)
        return 1;
    else
        return 3;
}();
```

Initialization

- And output parameters:

```
void get_value(int &v);
//...
const auto value = [] {
    int tmp = 0;
    get_value(tmp);
    return tmp;
}();
```

Temporary non-const variable in the scope of lambda only

Objects

- If accessed as const, ***this*** pointer is treated as pointer to const. That implies:
    - properties and bases are treated as const objects
    - only methods specified as “const” can be called

```
struct A {
    void doSmth() const;
    void doSmth();
};
const A a;
a.do_smth();
```

Objects

- But, const method can return non-const references and pointers:

```
class A {
public:
    int *getPtr() const { return a; }
    int &getRef() const { return *a; }
    std::shared_ptr<int> getSharedPtr() const { return b; }
private:
    int *a;
    std::shared_ptr<int> b;
};
```

```
Implies
int *const ptr;
const std::shared_ptr<int> b;
```

Objects

- The strategy should be defined and the design should follow it:

    1. propagation of immutability to the owned objects
    2. independent mutability of the owned objects
    3. immutable logical (observable) state (not the physical immutability)

Propagation of the immutability

- More restrictive const overloads for accessors:

```
class A {
public:
    const int *get_ptr() const { return a; }
    int *get_ptr() { return a; }

    const int& getRef() const { return *a; }
    int& getRef() { return *a; }

    std::shared_ptr<const int> getSharedPtr() const { return b; }
    std::shared_ptr<int> getSharedPtr() { return b; }
    //...
```

pointer to const in const methods

Propagation of the immutability

- More restrictive const overloads for accessors:

```
class A {
public:
    const int *get_ptr() const { return a; }
    int *get_ptr() { return a; }

    const int& getRef() const { return *a; }
    int& getRef() { return *a; }

    std::shared_ptr<const int> getSharedPtr() const { return b; }
    std::shared_ptr<int> getSharedPtr() { return b; }
    //...
```

Reference to const

Propagation of the immutability

- More restrictive const overloads for accessors:

```
class A {
public:
    const int *get_ptr() const { return a; }
    int *get_ptr() { return a; }

    const int& getRef() const { return *a; }
    int& getRef() { return *a; }

    std::shared_ptr<const int> getSharedPtr() const { return b; }
    std::shared_ptr<int> getSharedPtr() { return b; }
    //...
```

smart pointer to const in const methods

Independent mutability

- Owner is immutable but owned objects can be modified:

```
class Object {
public:
    Pool &getOwner();
};
class Pool {
public:
    std::shared_ptr<Object> getObject(Id id) const;
    void removeObject(Id id);
};
```

Independent mutability

- Owner is immutable but owned objects can be modified:

```
class Object {
public:
    Pool &getOwner();
};
class Pool {
public:
    std::shared_ptr<Object> getObject(Id id) const;
    void removeObject(Id id);
};
```

returns non-const reference/pointer to owner (can be provided in constructor)

Independent mutability

- Owner is immutable but owned objects can be modified:

```
class Object {
public:
    Pool &getOwner();
};
class Pool {
public:
    std::shared_ptr<Object> getObject(Id id) const;
    void removeObject(Id id);
};
const Pool pool;
pool.getObject(id)->getOwner().removeObject(id);
```

Enables non-const access to the owner

Independent mutability

- Owner is immutable but owned objects can be modified:

```
class Object {
public:
    Pool &getOwner();
};
class Pool {
public:
    std::shared_ptr<Object> getObject(Id id) const;
    void removeObject(Id id);
};
const Pool pool;
pool.getObject(id)->getOwner().removeObject(id);
```

Should be avoided

Immutable visible state

- Fields not visible externally are modified in const methods:

```
class CachedContainer {
    Item getItem(Id id) const {
        // if not available in cache
        const auto &it = container.find(id);
        cache.add(id, *it);
        return it;
    }

    Container container;
    mutable Cache cache;
};
```

Immutable visible state

- Fields not visible externally are modified in const methods:

```
class CachedContainer {
    Item getItem(Id id) const {
        // if not available in cache
        const auto &it = container.find(id);
        cache.add(id, *it);
        return it;
    }

    Container container;
    mutable Cache cache;
};
```

Field is marked as ***mutable***

Immutable visible state

- Fields not visible externally are modified in const methods:

```
class CachedContainer {
    Item getItem(Id id) const {
        // if not available in cache
        const auto &it = container.find(id);
        cache.add(id, *it);
        return it;
    }

    Container container;
    mutable Cache cache;
};
```

The const method modifies internal state not visible to client code

Immutable visible state

- Reflects logical immutability (immutability of the represented object)

```
struct IDatabase {
    virtual void addRecord(const Record &) = 0;
    virtual ConstRecordsIterator iterateRecords() const = 0;
    //...
};
```

Not real immutability: synchronization is required if methods are concurrently invoked (regardless of const specification);

Recommendations/considerations (reflect my opinion)

- Strictest levels of compiler diagnostics (and static analysis) should be used
- Objects should be const by default
- Const qualification for pointer objects (if have to be used) should not be ignored
- *auto* should be used for type deduction only and reference or pointer should be explicitly specified (with explicit const qualification)
- Consistent strategy for constness for owned objects should be maintained
- Mutability of the owned data in const methods should be used with caution
- As “const” is not equivalent to immutable, it can’t be assumed that object marked is const is safe in multithreaded environment

Thank you

Questions?