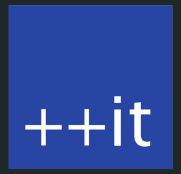
How to use `const` in C++

Sandor Dargo

The Italian C++ Conference

19th June, 2021



Italian C++ Community

www.italiancpp.org

Who Am I?

Sándor DARGÓ

- Software developer in Amadeus
- Enthusiastic blogger https://www.sandordargo.com
- (A former) Passionate traveller
- Curious home baker
- Happy father of two

Agenda

Why using `const` matters?

`const` local variables

`const` member variables

`const` functions

`const` return types

`const` parameters

Why using `const` matters?

Should we make everything `const`?

No!

But we should do better...

```
int MyRate::getNumberOfCoupons() {
  return _coupons.size();
}
```

```
std::string MessageFormatter::extractPhoneFromUnstructured(std::string unstructured) {
  std::string phone = "0000000000";
  try {
    size_t lastDigitIdx  = unstructured.find_last_of("0123456789");
    size_t firstDigitIdx = unstructured.find_last_not_of("0123456789()");
    // it takes last digits in the string and checks if the number is more than 10 digits
    if (lastDigitIdx && firstDigitIdx && (lastDigitIdx - firstDigitIdx > 9)) {
      if (lastDigitIdx - firstDigitIdx > 20) {
        phone = iUnstructured.substr(firstDigitIdx, 20);
      }
      else {
        phone = iUnstructured.substr(firstDigitIdx);
      }
    }
  }
  catch (...) {
    log("Error while extracting Phone!");
  }
  return phone;
}
```

```
bool TransactionCommit::checkIfSpecialException(CustomError& exception) {
  const std::string currentError = exception.what();
  const std::string specialException = "I'm so special";
  if (currentError.find(specialException) != std::string::npos) {
      return true;
  }
  return false;
}
```

Arguments against using `const`

Visual noise

Confuses the developer

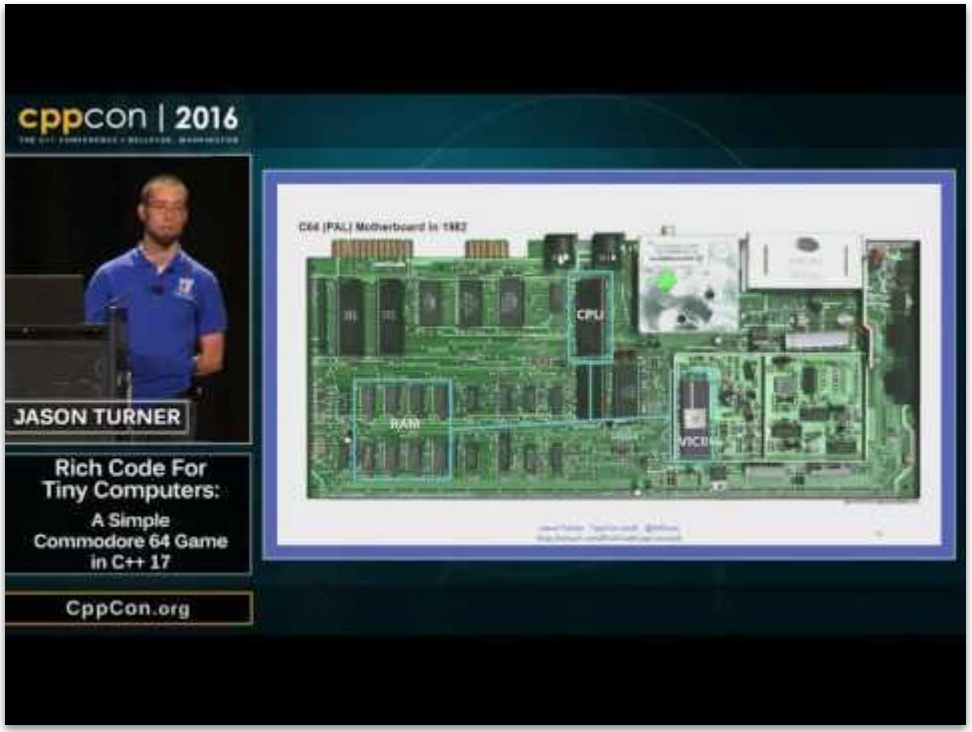
Doesn't matter

Arguments against using `const`

Visual noise

Confuses the developer

Doesn't matter



In reality, `const` helps because it...

Clarifies intent

- Should a variable be modified
- Should a function modify the internal state of an object

Makes code easier to understand

- For the readers
- For the compiler

It's so important that a new language was born out of the idea*:

`const` local variables

Let's create immutable variables

```
auto result = computeResult(); => const auto result = computeResult();
```

Shows clear intentions

- To the reader
- To the compiler

Protects against accidental changes

Default option in Rust and highly recommended in other languages

Con.1: By default, make objects immutable

Con.4: Use const to define objects with values that do not change after construction

What if you cannot simply initialize a variable?

Initial value might depend on a condition

Maybe a loop is used

There are solutions! You can use...

- a ternary
- a helper function
- an immediately invoked lambda (IIFE)
- algorithms

`const` member variables

Is that a good idea?

What would be the reason?

To show that a member will not change?

What will happen?

```cpp
class MyClass {
  const int m_num = 5;
};

int main() {
  MyClass c;
  MyClass c2;
  c = c2;
}
```

```
main.cpp: In function 'int main()':
main.cpp:11:7: error: use of deleted function 'MyClass& MyClass::operator=(const MyClass&)'
   11 |   c = c2;
      |       ^~
main.cpp:1:7: note: 'MyClass& MyClass::operator=(const MyClass&)' is implicitly deleted because the default definition would be ill-formed:
    1 | class MyClass {
      |       ^~~~~~~
main.cpp:1:7: error: non-static const member 'const int MyClass::m_num', cannot use default assignment operator
```

Having special functions is tricky!

How to implement assignment?

How to move away from a `const` member?

You need `const_cast` but that might lead to Undefined Behaviour*:

$5.2.11/7 - Note: Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_cast that casts away a const-qualifier may produce undefined behavior (7.1.5.1).

```
#include <utility>
#include <iostream>

class MyClassWithConstMember {
public:
  MyClassWithConstMember(int a) : m_a(a) {}
  MyClassWithConstMember& operator=(const MyClassWithConstMember& other) {
    int* tmp = const_cast<int*>(&m_a);
    *tmp = other.m_a;
    std::cout << "copy assignment \n";
    return *this;
  }

int getA() {return m_a;}

private:
  const int m_a;
};

int main() {
  MyClassWithConstMember o1{666};
  MyClassWithConstMember o2{42};
  std::cout << "o1.a: " << o1.getA() << std::endl;
  std::cout << "o2.a: " << o2.getA() << std::endl;
  o1 = o2;
  std::cout << "o1.a: " << o1.getA() << std::endl;
```

What to do instead?

Unless you really want to `const_cast` and UB...

- Keep members private*
- Don't expose "immutable" members via setters
- Keep functions `const` as much as possible
- Consider `static const` members if makes sense

When can `static const` make sense?

When the value cannot change among instances

When the value cannot change during the whole lifetime of the program

=> When you need a (global) constant*

`const` members are for object lifetime, `static const`s are for program lifetime

`const` functions

Use them without moderation!

A `const` function cannot change the underlying object

(It can change other objects)

Conveys a message

It's a guarantee both to the reader and a compiler

Use it whenever you don't (want to) modify the underlying object

Con.2: By default, make member functions const

What's the situations with overloads?

What happens when you have a `const` and a non-`const` overload?

```cpp
#include <iostream>

class MyClass {
  public:
  void foo() const {
      std::cout << "void MyClass::foo() const is called\n";
  }

  void foo() {
      std::cout << "void MyClass::foo() non-const is called\n";
  }
};

int main() {
  MyClass o;
  const MyClass o2;
  o.foo();
  o2.foo();
}
```

Compiler will choose based on whether the object is `const`

```
void MyClass::foo() non-const is called
void MyClass::foo() const is called
```

What if I have a non-`const` instance?

```
((const decltype(o)) o).foo();
```

Ok, that was a joke ;)

```
static_cast<const MyClass>(o).foo(); // pre-C++17
```

```
std::as_const(o).foo();
```

`const` return types

Damn it, that's dangling!

Be cautious with `const` references!

- Never return local objects by reference as they get destroyed
- You must make sure that you don't use the returned object outside of the `MyObject`'s instance scope

```
const T& MyObject::getSomethingConstRef() {
  T ret;
  // ...
  return ret; // ret gets destroyed right after, the returned reference points at its ashes
}
```

```
class MyObject
{
public:
  // ...
  const T& getSomethingConstRef() {
    return m_t; // m_t lives as long as our MyObject instance is alive
  }
private:
  T m_t;
};
```

"All that glitters is not gold"

What about returning `const` value objects?

What intention `const std::string foo()` shows?

- Value: the caller gets a copy
- `const`: the returned object shouldn't be modified

Does that make sense?

Is it misleading?

```cpp
#include <iostream>
#include <string>

const std::string foo() {
  return std::string{"bar"};
}

int main() {
  auto s = foo();
  s += "baz";
  std::cout << s << '\n';
}
```

```
barbaz

g++ -std=c++20 -Wall -pedantic
```

Can returning a const value decrease performance?

Can we pass `const SgWithMove` as `SgWithMove&&`?

It would discard the `const` qualifier

Fallback to a silent copy

```cpp
#include <utility>
#include <iostream>

class SgWithMove{
    public:
    SgWithMove() = default;
    ~SgWithMove() = default;

    SgWithMove(const SgWithMove&) = default;
    SgWithMove(SgWithMove&&)      = default;

    SgWithMove& operator=(const SgWithMove&) {
        std::cout << "copy assign\n" ;
        return *this;
    }
    SgWithMove& operator=(SgWithMove&&) {
        std::cout << "move assign\n" ;
        return *this;
    }
};

const SgWithMove foo() {
    return SgWithMove{};
}

int main() {
    SgWithMove o;
    o = std::move(foo());
}
```

Pointers are similar to references but "worse"

Never return local objects created on the stack by their address to avoid dangling pointers...

Ensure proper life-times

But there is more!

But before... east `const` vs `const` west

East `const`

What's left of `const` is constant

```
int const c = 1;

int const& cr = c;

int const* pc = &c; // pointer to
 const int

int *const cp = &c; // const
 pointer to int

int const*const cpc = &c; // const
 pointer to const int
```

`const` west

More widespread but less consistent

```
const int c = 1;

const int& cr = c;

const int* pc = &c; // pointer to
 const int

int *const cp = &c; // const
 pointer to int

const int *const cpc = &c; //
 const pointer to const int
```

`int * const func() const`

Function is `const`

The returned pointer is `const`, but the data we point to can be modified

Type qualifiers are ignored on function return types, the return type is a pointer

`const` is ignored!

```cpp
#include <iostream>

class A {
public:
    int * const func () const {
        int * a = new int{42};
        return a;
    }
};

int main() {
    A a;
    auto* num = a.func();
    std::cout << *num << '\n';
    ++(*num);
    std::cout << *num << '\n';
    num = new int{666};
    std::cout << *num << '\n';
}
```

```
main.cpp:5:5: warning: type qualifiers ignored on function return type [-Wignored-qualifiers]
    5 |     int * const func () const {
      |     ^~~
42
43
666
```

`const int * func () const`

Function is `const`

The returned pointer can be changed, but the data is `const`

Constness is taken into account

Hence compilation fails

```cpp
#include <iostream>

class A {
public:
    const int * func () const {
        int * a = new int{42};
        return a;
    }
};

int main() {
    A a;
    auto * num = a.func();
    std::cout << " " << *num << '\n';
    ++(*num);
    std::cout << " " << *num << '\n';
    num = new int{666};
    std::cout << " " << *num << '\n';
}
```

```
main.cpp: In function 'int main()':
main.cpp:15:8: error: increment of read-only location '* num'
   15 |     ++(*num);
      |       ~^~~~~
```

const int * const func() const

Similar behaviour!

- `func()` is `const`
- The returned pointer can be changed, but the data is `const`
- A compiler warning on ignored qualifiers
- An applied constness on the data

```cpp
#include <iostream>

class A {
public:
  const int * const func() const {
    int * a = new int{42};
    return a;
  }
};

int main() {
  A a;
  auto * num = a.func();
  std::cout << " " << *num << '\n';
//   ++(*num);
//   std::cout << " " << *num << '\n';
  num = new int{666};
  std::cout << " " << *num << '\n';
}
```

```
main.cpp:5:3: warning: type qualifiers ignored on function return type [-Wignored-qualifiers]
    5 |   const int * const func() const {
      |   ^~~~~
 42
 666
```

`const` parameters

Non-reference type parameters' constness can be ignored

Don't use this "feature"!

- `const` can be ignored when you try to overload a value type or implement a function
- It's very misleading
- For references and pointers, `const` is not ignored

```
1  #include <iostream>
2
3  class A {
4  public:
5    void foo(const int bar);
6  };
7
8  void A::foo(int bar) {
9    std::cout << bar << '\n';
10   bar++;
11   std::cout << bar << '\n';
12 }
13
14 int main() {
15   A a;
16   a.foo(42);
17 }
```

`const` class type parameters!

And reference as a rule of thumb!

Copying an object is more expensive than passing a reference*

Take object parameters as `const&` by default

If you'd later have to modify it but not the original, you can take it simply by value

Though you might wonder as a reader if it was on purpose...

Con.3: By default, pass pointers and references to consts

```
void doSomething(const ClassA& foo) {
// ...
ClassA foo2 = foo;
foo2.modify();
// ...
}
```

```
void doSomething(ClassA foo) {
// ...
foo.modify();
// ...
}
```

Conclusion

Using const is useful...

...just know the rules!

Member variables: rather not

Functions: without moderation!

Local variables: whenever possible!

Return types: be cautious and avoid lifetime issues!

Parameters: if it doesn’t make you create another copy

Benefit from `const` to

Improve maintainability

Improve const correctness

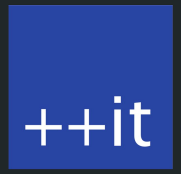
Use it as an exploration tool

How to use `const` in C++

Sandor Dargo

The Italian C++ Conference

19th June, 2021



Italian C++ Community

www.italiancpp.org