How to use `const` in C++

Sandor Dargo

Agenda

Why using `const` matters?

`const` local variables

`const` member variables

What is `constexpr`?

`const` functions

`const` return types

`const` parameters

`const` and smart pointers

`const` and rvalue references

`const` and templates

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

Who are you?

Something personal

Something professional

Something you expect

Why using `const` matters?

Should we make everything `const`?

No!

But we should do better...

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

```cpp
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;
}
```

```cpp
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

Is `const` is visual noise?

```
auto numberOfDoors{2u};
```

or

```
const auto numberOfDoors{2u};
```

Comments as visual noise!

```
class Car {
public:
  // ...
  /*
  * Gets the performance in horsepower
  */
  std::string getHorsepower() {
    return m_horsePower;
  }
private:
  int m_horsePower; // Performance in horse power
};
```

Duplicated type information

```
int* myInt = new int{42};

auto* myOtherInt = new int{42};


unsigned int num{42u};

auto otherNum{42u};
```

Things that could have been done simpler....

Overcomplicated “smart” code

Raw loops instead of algorithms

Etc.

Is `const` confusing for the developer?

It’s not smart code

It reveals intention to the reader and to the compiler

How can it confuse anybody?

By its lack!

What if you have a codebase without `const`?

- Will you add it everywhere?
- Or only for new code?
- Or not at all?

Don't follow a foolish consistency!

What is foolish consistency?

*“**A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.** With consistency a great soul has simply nothing to do. He may as well concern himself with his shadow on the wall. Speak what you think now in hard words, and tomorrow speak what tomorrow thinks in hard words again, though it contradict every thing you said today. — 'Ah, so you shall be sure to be misunderstood.' — Is it so bad, then, to be misunderstood? Pythagoras was misunderstood, and Socrates, and Jesus, and Luther, and Copernicus, and Galileo, and Newton, and every pure and wise spirit that ever took flesh. To be great is to be misunderstood.”* - Ralph Waldo Emerson, Self-Reliance

Then Jon Kalb came

<< “A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.” I don’t think he was talking about code, but that statement couldn’t be more relevant to software engineers. >> -

http://slashslash.info/2018/02/a-foolish-consistency/

But what is foolish consistency after all?

- Avoiding something better, because we cannot update everything
  - Yet, accidental inconsistencies are still bad
  - Improvement should be embraced, it’s called growth
- Staying consistent means never improving

It doesn’t matter anyway… does it?

It does!

And at least it won’t hurt

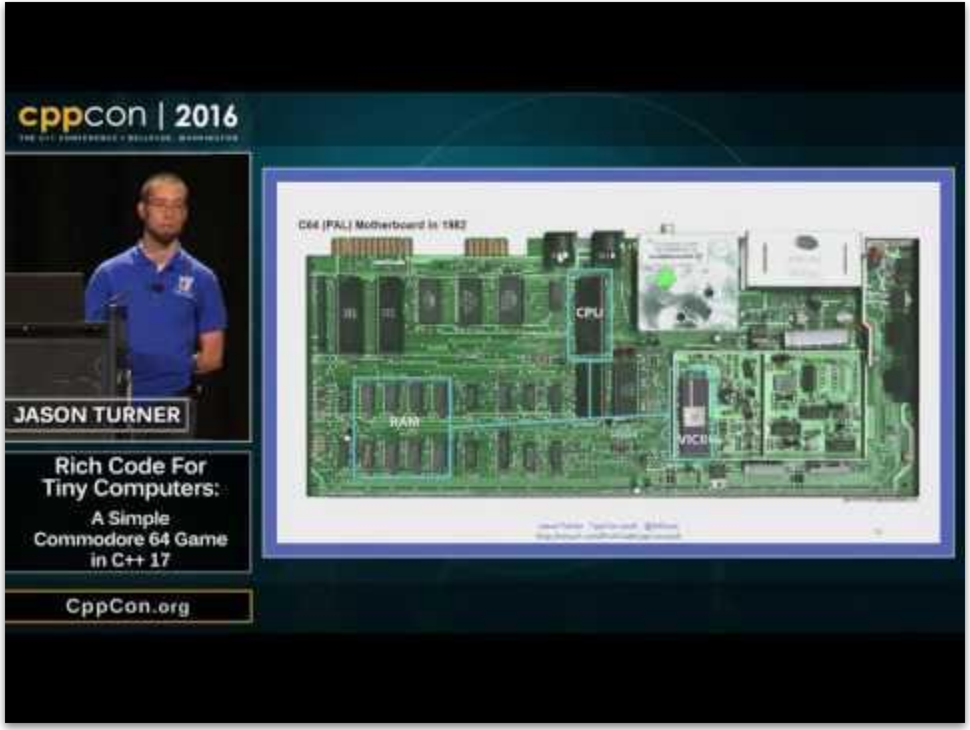
If used correctly…

It doesn't matter anyway... does it?

It does!

And at least it won't hurt

If used correctly...



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*:

The Rust Programming Language

Using `const` well leads to `const` correct code

<u>Use `const` to prevent `const` objects from getting mutated</u>

No accidental modifications -> a type of *type safety*

The sooner the better

If later, don't do it all at once

You can use it with trial and fail

`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

Initialize a variable `const` with a ternary

When there is only possible value:

```
int foo = bar(); => const int foo = bar();
```

When 2 values:

```
int foo;
if (baz) { foo = bar(); }
else { foo = blah(); }
```

=>

```
const int foo = baz ? bar() : blah();
```

What if there are several ways to initialize a variable?

Initialization often with

- 😞 a long if else
- 😞 a switch

DO NOT USE NESTED TERNARIES!

```
int foo;

if (cond1) { foo = bar(); }

else if (cond2) {foo =
baz();}

else {foo = blah()};
```

Initialize a variable `const` with a helper

```
int initFoo() {
  if (cond1) { return bar(); }
  else if (cond2) {return baz();}
  else {return blah()};
}
const int foo = initFoo();
```

Initialize a variable `const` with IIFE

```
const int foo = [&](){
  if (cond1) { return bar(); }
  else if (cond2) {return baz();}
  else {return blah();};
}();
```

[Let’s have a look at this at C++ Insights]

What is Return Value Optimization?

Don’t copy local variables to be returned into the return variables if

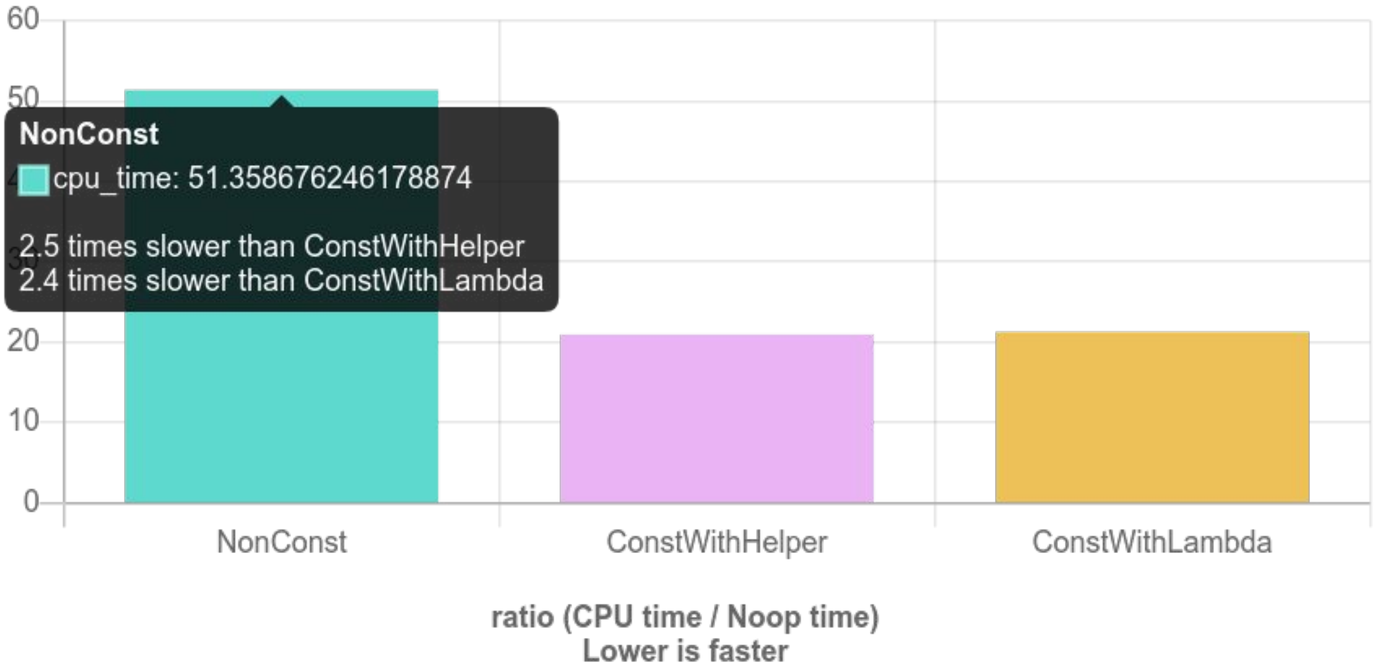
- type of local object == type of return type

- the local object is returned

```
Widget makeWidget() {
  Widget w;
  // …
  // "copy" w into return value
  return w;
}
```

What about performance?

Quickbench is our friend!



What if you modify your variable in a loop?

Initialize-then-modify anti pattern

First you declare and initialize

Later you modify usually in a loop

Great talks on this by

- Sean Parent
- Ben Deane
- Conor Hoekstra

```
std::vector nums{1, 51, 43,
                 42, 89};
bool ret = false;
for (auto n : nums) {
    if (n % 2 == 0) {
        ret = true;
        break;
    }
}
```

Use an algorithm instead

Most for loops can be replaced by an algorithm

They are more expressive

They are often more performant

They are less error-prone

```
const bool found =
  std::any_of(nums.begin(),
              nums.end(),
              [](auto n){
                return n % 2 == 0;
              });
```

Get rid of raw loops and make the result `const`

```
std::vector nums{1, 2, 3, 4, 5};

auto r = 0;

for(auto n: nums) {

  r += n;

}
```

```
std::vector nums{1, 2, 3, 4, 5};

const auto sum =
  std::accumulate(nums.begin(),
                  nums.end(), 0);
```

Get rid off raw loops and make the result `const`

```
std::vector nums{1, 2, 3, 4, 5};


auto r = 0;

for (auto n: nums) {

  if (n % 2 == 0) {

    ++r;

  }

}
```

```
std::vector nums{1, 2, 3, 4, 5};

const auto count =
    std::count_if(
        nums.begin(),
        nums.end(),
        [](auto n){
            return n % 2 == 0;
    });
```

Get rid off raw loops and make the result `const`

```
std::vector nums{1, 2, 3, 4, 5};

auto r = std::end(nums);
for (auto n=nums.begin();
      n!=nums.end(); ++n) {
  if (*n % 2 == 0) {
      r = n;
      break;
  }
}
```

```
std::vector nums{1, 2, 3, 4, 5};


auto match =
  std::find_if(
    nums.begin(),
    nums.end(),
    [](int n){
      return n % 2 == 0;
  });
```

`const std::vector` vs `constexpr std::array`

Dynamic array, size cannot change though

Size can be calculated at run-time

Static array, so size cannot change

Size must be known at compile-time

More efficient

Choose a `std::array` whenever you know the full content at compile-time!

`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 is (observable) behaviour of code?

Guaranteed behaviour

Ill-formed

Ill-formed no diagnostic required

Unspecified behaviour

Implementation defined behaviour

Undefined behaviour

Unspecified behaviour

Rules are not specified by the Standard

Implementation doesn’t have to document

Different result sets are valid

No crash, strictly limited perimeters

Examples:

&x > &y

expression evaluation order

```cpp
#include <iostream>
int x=333;

int add(int i, int j) {return i+j;}
int left() {
  x = 100;
  return x;
}
int right() {
  x++;
  return x;
}
int main() {
 std::cout << add(left(), right()) << std::endl;
}
```

```
434
```

```
g++ -std=c++14 -Wall -pedantic -pthread mai
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
› clang++-7 -pthread -o main main.cpp
› ./main
201
›
```

Implementation defined behaviour

Like unspecified behaviour

But implementation must document the result

Examples:

- Default integer types
- Number of bits in a byte
- ORDER BY on NULLs

Undefined behaviour

We break the rules, no requirements on the behaviour

Anything can happen to the entire program, the compiler owes us nothing

- Crash
- Logically impossible results
- Non-deterministic behaviour
- Removed execution paths

Examples

- Accessing uninitialized variables
- Deleting object through base class pointer w/o virtual destructor

Why does UB exist?

Portability

Performance optimizations

Make APIs shorter

Simpler implementations

Let's talk about `const_cast`

To remove the constant nature of any object

To pass a constant object to a non-const API

To change the value of something referenced as `const`

Yet, you cannot change a `const` value through a non-`const` access path

It's a bad practice!

`const_cast` or not to cast?

```
#include <iostream>

int main()

{

  int number = 3;                // number is not declared
const

  const int&  const_ref_number = number;

  std::cout << "old number = " << number << '\n';

  const_cast<int&>(const_ref_number) = 4; // OK:
modifies number

  std::cout << "new number = " << number << '\n';

}
```

```
#include <iostream>

int main()

{

  const int number = 3;                // number is declared
const

  const int&  const_ref_number = number;

  std::cout << "old number = " << number << '\n';

  const_cast<int&>(const_ref_number) = 4; // UB:
modifies number

  std::cout << "new number = " << number << '\n';

}
```

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

`static const` or `static constexpr`?

What is `constexpr`?

What are “constant expressions”?

Initializable at compile time

Thread safe

Can be stored in ROM

No run-time costs

constant expressions can be used as

Non-type template arguments

Array sizes

Etc.

`constexpr` variables

Are implicitly const

Have to be initialized

Can only be initialized by a constant expression

`constexpr` functions

Must not be `virtual` (until C++20), lately not `coroutine`

Are implicitly `const` only in C++11

Have quite restricted bodies

`constexpr` functions in C++11

Have quite restricted bodies

Non-virtual

Literal arguments and return value

Body must be defaulted/deleted or it can be a one return statements

You can use ternaries and recursion!

What is a literal type?

Scalar

Reference

Array of literal

A cv qualified class

- With a trivial (until C++20), then a `constexpr` destructor
- With an Aggregate/clojure/`constexpr` constructor
- Data members/base classes are literal, non-`volatile` (until C++17)
- For unions, at least one non-static member is non-`volatile`, literal
- For non-unions, all non-`static` members and bases are non-`volatile` literals

Simple `constexpr` function in C++11

```
#include <iostream>

constexpr int power(int num){
  return num * num;
}

int main(){
  constexpr int i= power(42);
  int num = 66;
  int powerOfNum = power(num);

  std::cout << "power(42): " << i
<< '\n';
  std::cout << "power(num): " << 
powerOfNum << '\n';
}
```

power(42): 1764

power(num): 4356

Recursive `constexpr` in C++11

```
#include <iostream>

constexpr int fibonacci(int n) {
  return (n == 0 || n == 1)
      ? n : fibonacci(n-1) +
            fibonacci(n-2);
}

int main(){
  constexpr int fibo10=
      fibonacci(10);
  std::cout << "fibonacci(10): "
            << fibo10 << '\n'
}
```

0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	**55**

`constexpr` functions in C++14

Conditional jump instructions or loop instructions

More than one instruction

`constexpr` functions

Fundamental data types initialized with a constant expression

No static, no thread-local, no try-catch, no goto

Recursive multi-expression `constexpr` in C++14

```
#include <iostream>

constexpr int fibonacci(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fibonacci(n-1) +
           fibonacci(n-2);
  }
}

int main(){
  constexpr int fibo10 = fibonacci(10);
  std::cout << "fibonacci(10): "
            << fibo10 << '\n'
}
```

0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	**55**

`constexpr` function with a loop in C++14

```
#include <iostream>

constexpr int fibonacci(int n) {
  int i=0, next=0, first=0, second=1;
  while (i <= n) {
    if (i < 2) {
      next = i;
    } else {
      next = first + second;
      first = second;
      second = next;
    }
    ++i;
  }
  return next;
}

int main(){
  constexpr int fibo10 = fibonacci(10);
  std::cout << "fibonacci(10): "
            << fibo10 << '\n'
}
```

0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	**55**

`constexpr` functions in C++20

Can be `virtual`, but not `coroutine`

Can have a try-catch block

Can have an asm declaration

Can have variables without initialization (DEMO!)

`constexpr` functions and purity

`constexpr` functions can be executed at runtime

Compile-time has no state => function is pure

Pure functions

- Always return the same results for the same arguments
- Have no side effects
- Easy reordering and refactoring

`static const` or `constexpr`?

Use `constexpr` for anything that is known at compile time

`static const` might be runtime -> possible penalty

No out of line definition for `static constexpr` since C++17

Don't omit the the `static` from `constexpr`

The compiler could decide to set the value later

Clearer intentions for `constexpr`, prefer that one!

`const` functions

Use them without moderation!

A `const` function cannot change the underlying object

No member can be modified

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();
```

Is a `constexpr` function implicitly `const`?

Would this compile?

It depends!

For C++11, `constexpr` functions are implicitly `const`

From C++14, no implicit `const`ness

```
class A {
public:
  constexpr A(int a) : m_a(a)
{ }
  constexpr int fortytwo() {
    m_a = 42;
    return m_a;
  }
private:
  int m_a;
};

int main() {
  A a{5};
  a.fortytwo();
}
```

`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;
};
```

Non-`const` reference from `const` member function?

The idea seems bad…

The caller could change the Person’s name

Luckily (modern) compilers fail!

error: binding reference of type 'std::string&' {aka 'std::__cxx11::basic_string<char>&'} to 'const string' {aka 'const std::__cxx11::basic_string<char>'} discards qualifiers

```cpp
class Person {
public:
  Person(std::string name)
     : m_name(name) {}
  std::string& name() const {
     return m_name;
  }
private:
  std::string m_name;
};
```

"All that glitters is not gold"

What about returning `const` value objects?

What intention `const std::string getName()` communicates?

- `std::string` (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&&`?

```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());
}
```

What are move semantics?

Transferring content between objects

Can replace costly copy operations

Leaves the source “empty”

```
std::string bottle("full");

std::cout << "bottle is " << bottle
<< ".\n";

std::string
jug(std::move(bottle));

std::cout << "jug is " << jug
          << ", and the bottle is"
          << bottle << ".\n";
```

bottle is full.

jug is full, and the bottle is.

When is it possible to move?

Passing an object to a function

```
std::string bar = "bar-string";
std::vector<std::string> myvector;
myvector.push_back (std::move(bar));
```

Returning an object from a function

```
template <typename T> T&& min_(T&& a, T &&b) {
    return std::move(a < b? a: b);
}
```

If the object is:

- An rvalue
- The object’s class defines special member functions

What are rvalue references

an lvalue is an expression whose address can be taken, a locator value. Anything you can make assignments to is an lvalue

an rvalue is an unnamed value that exists only during the evaluation of an expression

`&&` operator denotes rvalue references (introduced in C++11)

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 = 1;

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

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

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

`const` west

More widespread but less consistent

```
const int c = 1;

const int& cr = 1;

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

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

const int *const cpc = &i; //
 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

Semantically the same

- `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

`const` primitives?

For sure not as `const` references

- Reference adds indirection
- Less efficient memory read

But what about `const int`?

It depends...

- Can mark an intention (don't change the value!)
- But don't make it `const`, if you need a copy later anyway

```
void setFoo(const int foo) {
  this->m_foo = foo;
}
```

```
void doSomething(const int foo) {
// ...
int foo2 = foo;
foo2++;
// ...
}
```

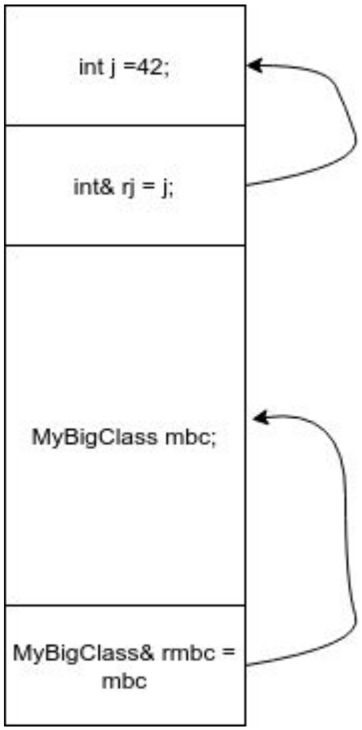
What happens when you pass by reference?

You pass an address

If your type is small, it’s not worth it

It gives an extra jump

If the type is big, it’s cheaper than a copy



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

```cpp
#include <iostream>

class A {
public:
  void foo(const int bar);
};

void A::foo(int bar) {
  std::cout << bar << '\n';
  bar++;
  std::cout << bar << '\n';
}

int main() {
  A a;
  a.foo(42);
}
```

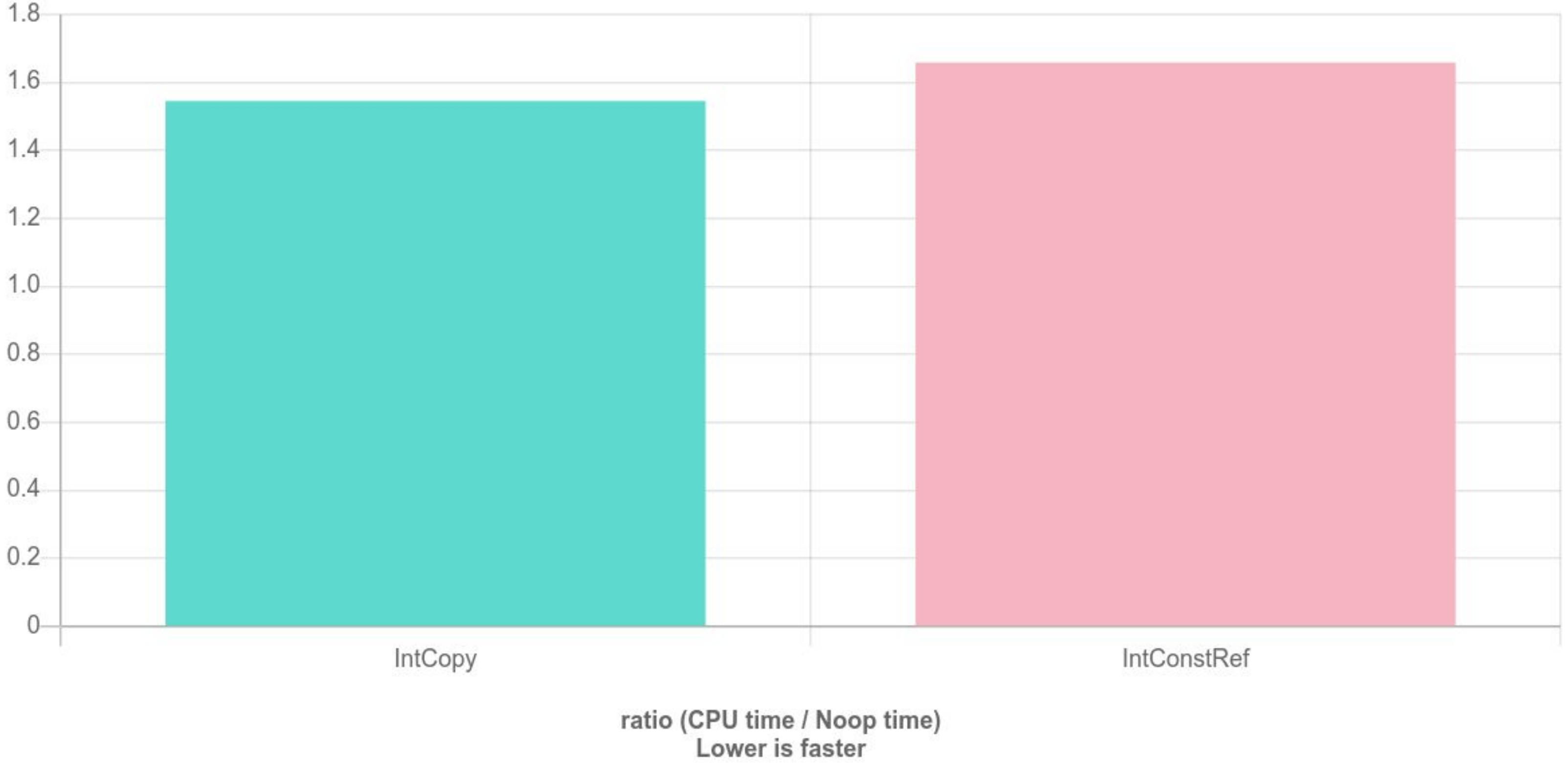
The other way around if might the implementer

Adding `const` to the definition means a hint to the readers and maintainers of the code

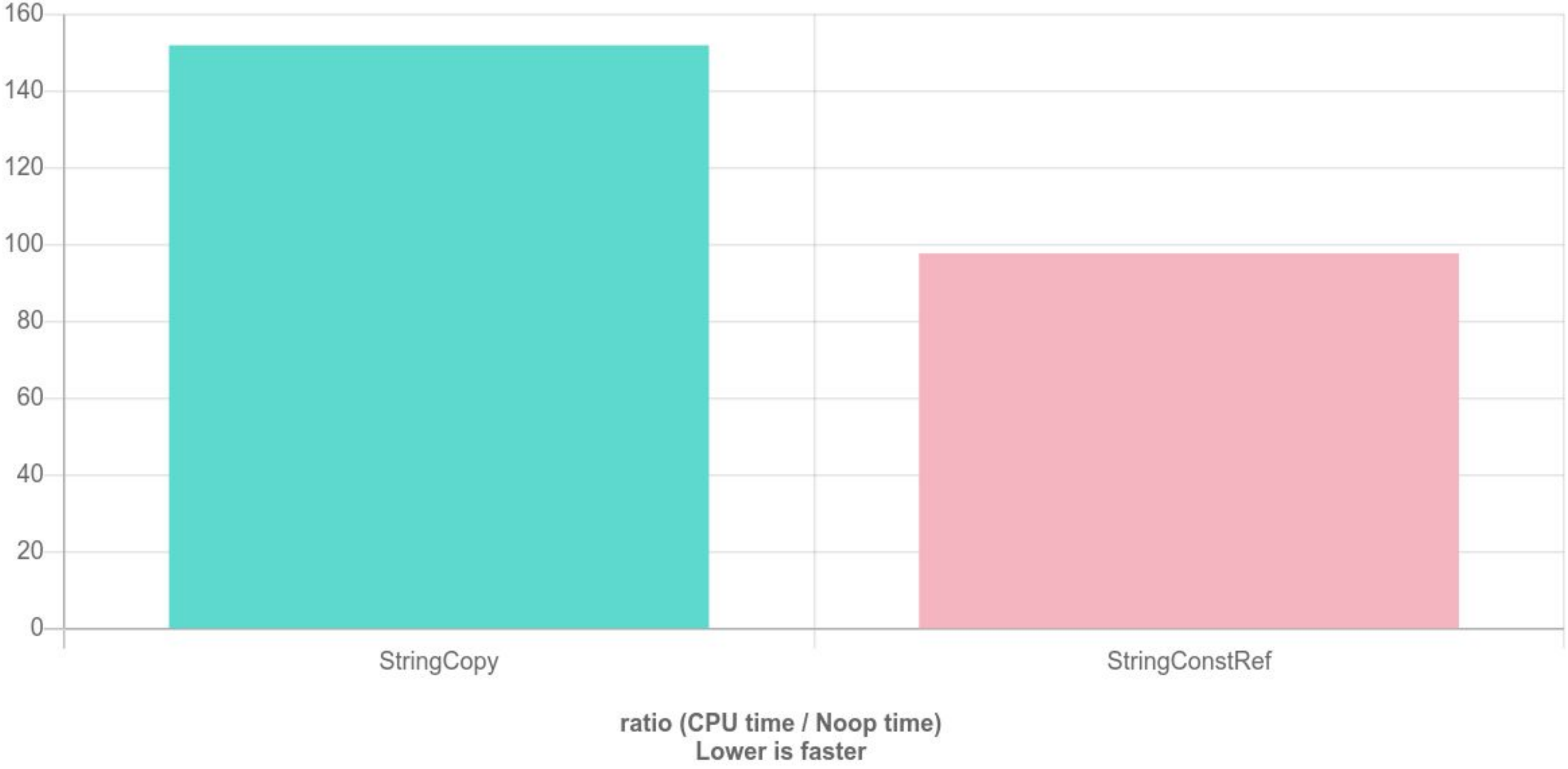
It's not misleading as it doesn't weaken the communicated contract

```
#include <iostream>

class A {
public:
  void foo(int bar);
};

void A::foo(const int bar) {
  // ++bar; // oh I cannot do
this!
}

int main() {
  A a;
  a.foo(42);
}
```

Copying a primitive is slower than taking by reference



Referencing a string is faster than copying it



`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

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

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...

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

[Con.3: By default, pass pointers and references to consts]()

Use `string_view` instead of `const& string` (C++17)

A string_view is a non-owning reference to a string

It's typically composed of

- A pointer to a character sequence
- Length

Cheap to copy!

A `string_view` is a good replacement for

Read operations

Some simple modifications

- Remove prefix
- Remove suffix

```cpp
#include <cstring>
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "balloon" };

  // Remove the "b"
  str.remove_prefix(1);
  // remove the "oon"
  str.remove_suffix(3);
  // Remember that the above doesn't modify the string, it only changes
  // the region that str is observing.

  std::cout << str << " has " << std::strlen(str.data()) << " letter(s)\n";
  std::cout << "str.data() is " << str.data() << '\n';
  std::cout << "str is " << str << '\n';

  return 0;
}
```

```
all has 6 letter(s)
str.data() is alloon
str is all
```

Say goodbye to `const& std::string`

Replaces const char* and std::string APIs by one common API

When no need for ownership

Yet comes with a decent API

Can bring significant performance gain

`consts` and smart pointers

What are smart pointers?

Smart pointers enable

automatic

exception-safe

object lifetime management

Smart pointers are objects themselves

They wrap pointers

Make sure that pointer is deleted when it goes out of scope

So how can it be const?

const std::shared_ptr<T>

The data is not const, so it can change

The place of memory it points to is const, it cannot change

```
const std::shared_ptr<MyInt> pn =
    std::make_shared<MyInt>(5);
++(*pn);
std::cout << *pn << '\n'; // 6
// pn.reset(new MyInt(6)); // ERROR
```

std::shared_ptr<**const** T>

The data is const

The place of memory it points to is not const

Having const in std::make_shared is not mandatory, but it avoid extra conversions and copies

```
std::shared_ptr<const MyInt> pn =
    std::make_shared< const
MyInt>(5);
// ++(*pn); // ERROR
std::cout << *pn << '\n'; // 5
pn.reset(new MyInt(6));
std::cout << *pn << '\n'; // 6
```

const std::shared_ptr<const T>

The data is const

The place of memory it points to is also const

```
const std::shared_ptr<const MyInt>
pn =
    std::make_shared< const
MyInt>(5);
// ++(*pn); // ERROR
std::cout << *pn << '\n'; // 5
// pn.reset(new MyInt(6)); // ERROR
```

But what about `const&` smart pointers?

No change in ownership

Passing a reference to a smart pointer means that we don't pass the ownership

For shared_pointers the reference counter is not incremented

It's still the original caller who is responsible for destroying the object

Might lead to a dangling pointer

```
#include <iostream>
#include <memory>

void foosmart(std::shared_ptr<int> sp) {
    std::cout << "counter: " << sp.use_count()
              << std::endl;
}

void foosmartref(const std::shared_ptr<int>& spr) {
    std::cout << "counter: " << spr.use_count()
              << std::endl;
}

int main() {
  std::shared_ptr<int> p = std::make_shared<int>(5);
  foosmart(p);
  foosmartref(p);
}

/*
counter: 2
counter: 1
*/
```

Seems easier, but don’t get deceived

It’s still the original caller who is responsible for destroying the object

For unique_ptr, no need to use std::move

Too complex, too much thinking

```
#include <iostream>
#include <memory>

void foosmart(std::unique_ptr<int> sp) {
    std::cout<< *sp << std::endl;
}

void foosmartref(const std::unique_ptr<int>& spr) {
    std::cout<< *spr << std::endl;
}

int main() {
  std::unique_ptr<int> p = std::make_unique<int>(5);
  foosmartref(p);
  foosmartref(p);
  foosmart(std::move(p));
  //  foosmart(std::move(p)); // ERROR: Segmentation fault
}

/*
5
5
5
*/
```

Use just a bare pointer instead of a reference

Too complex, too much thinking

Use just a bare pointer instead, a const one if needed

```
#include <iostream>
#include <memory>

void foobare(int*const p) {
  // p = new int{*p + 1 }; // ERROR: assignment of read-only
  std::cout<< *p << std::endl;
}

void foosmart(std::unique_ptr<int> sp) {
  std::cout<< *sp << std::endl;
}

int main() {
  std::unique_ptr<int> p = std::make_unique<int>(5);
  foobare(p.get());
  foobare(p.get());
}

/*
5
5
*/
```

`consts` rvalue references

What are rvalue references?

an lvalue is an expression whose address can be taken, a locator value. Anything you can make assignments to is an lvalue

an rvalue is an unnamed value that exists only during the evaluation of an expression

`&&` operator denotes rvalue references (introduced in C++11)

“The main purpose of rvalue references is to allow us to move objects instead of copying them.”

Non-const by default

Moving implies modification

Move constructor and assignment operators are taking non-const rvalue references by default

```
class MyClass {
public:
  MyClass(const MyClass&) noexcept;
  MyClass(MyClass&&) noexcept;
  MyClass& operator=(const MyClass&)
noexcept;
  MyClass& operator=(MyClass&&) noexcept;
  virtual ~MyClass() noexcept;
};
```

Does `const T&&` exist?

Yes!

Binding rules are well-defined

```
#include <iostream>

struct S {};

void f (S&) { std::cout << "lvalue ref\n"; }  // #1
void f (const S&) { std::cout << "const lvalue ref\n"; }  // #2
void f (S&&) { std::cout << "rvalue ref\n"; } // #3
void f (const S&&) { std::cout << "const rvalue ref\n"; } // #4
const S g () {return S{};};

int main() {
    S x;
    const S cx;
    f (S{}); // rvalue        #3, #4, #2
    f (g()); // const rvalue  #4, #2
    f (x);    // lvalue        #1, #2
    f (cx);   // const lvalue  #2
}
```

Back to the your house, your palace metaphor

Why place any restrictions on what a caller of our function can do with the their own copy of the returned value?

Why try to move an immutable object?

When to use them?

Disallow rvalue references altogether in a bulletproof way

Disallow binding lvalue to rvalue

`void f (S&&) = delete;` only disallows rvalues

`void f (const S&&) = delete;` disallows both rvalues and `const` rvalues

`const` and templates

How can `const` appear in templates?

What if you want to compare types?

Base types?

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<L, R>();
}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // false
  std::cout << is_same_base<int, int*>() << '\n'; // false
  std::cout << is_same_base<int, const int*>() << '\n'; // false
  std::cout << is_same_base<int, int&>() << '\n'; // false
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

`std::remove_const` removes constness

`std::remove_cv` also removes `volatile`

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_const_t<L>,
                      std::remove_const_t<R>>();
}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // true
  std::cout << is_same_base<int, int*>() << '\n'; // false
  std::cout << is_same_base<int, const int*>() << '\n'; // false
  std::cout << is_same_base<int, int&>() << '\n'; // false
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

`std::remove_pointer` removes the pointer

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_pointer_t<L>,
                      std::remove_pointer_t<R>>();
}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // false
  std::cout << is_same_base<int, int*>() << '\n'; // true
  std::cout << is_same_base<int, const int*>() << '\n'; //false
  std::cout << is_same_base<int, int&>() << '\n'; // false
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

These meta functions are composable

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_const_t<std::remove_pointer_t<L>>,
                      std::remove_const_t<std::remove_pointer_t<R>>>();
}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // true
  std::cout << is_same_base<int, int*>() << '\n'; // true
  std::cout << is_same_base<int, const int*>() << '\n'; // true
  std::cout << is_same_base<int, int&>() << '\n'; // false
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

std::remove_reference removes reference

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_reference_t<L>,
                      std::remove_reference_t<R>>();}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // false
  std::cout << is_same_base<int, int*>() << '\n'; // false
  std::cout << is_same_base<int, const int*>() << '\n'; // false
  std::cout << is_same_base<int, int&>() << '\n'; // true
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

`std::decay` removes... many things

r and lvalue references

cv qualifiers

Converts functions to pointers to base type

Converts arrays to pointers

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_reference_t<L>,
                      std::remove_reference_t<R>>();}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // true
  std::cout << is_same_base<int, int*>() << '\n'; // false
  std::cout << is_same_base<int, const int*>() << '\n'; // false
  std::cout << is_same_base<int, int&>() << '\n'; // true
  std::cout << is_same_base<int, const int&>() << '\n'; // true
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

sStd::remove_cvref (C++20) removes...

cv qualifiers

And references

```
#include <iostream>
#include <type_traits>

template<typename L, typename R>
bool is_same_base() {
  return std::is_same<std::remove_cvref_t<L>,
                      std::remove_cvref_t<R>>();}

int main() {
  std::cout << std::boolalpha;
  std::cout << is_same_base<int, const int>() << '\n'; // true
  std::cout << is_same_base<int, int*>() << '\n'; // true
  std::cout << is_same_base<int, const int*>() << '\n'; // true
  std::cout << is_same_base<int, int&>() << '\n'; // true
  std::cout << is_same_base<int, const int&>() << '\n'; // false
  std::cout << is_same_base<int, float>() << '\n'; // false
}
```

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 const correctness

Use it as an exploration tool

How to use `const` in C++

Sandor Dargo