

Const Correctness



Original from : Mike Precup (mprecup@cs.stanford.edu)
ENJMIN Edition 2016

Why Const?

"I still sometimes come across programmers who think const isn't worth the trouble. "Aw, const is a pain to write everywhere," I've heard some complain. "If I use it in one place, I have to use it all the time. And anyway, other people skip it, and their programs work fine. Some of the libraries that I use aren't const-correct either. Is const worth it?"

We could imagine a similar scene, this time at a rifle range: "Aw, this gun's safety is a pain to set all the time. And anyway, some other people don't use it either, and some of them haven't shot their own feet off..."

Safety-incorrect riflemen are not long for this world. Nor are const-incorrect programmers, carpenters who don't have time for hard-hats, and electricians who don't have time to identify the live wire. **There is no excuse for ignoring the safety mechanisms provided with a product, and there is particularly no excuse for programmers too lazy to write const-correct code."**

- Herb Sutter, generally cool dude

Why Const?

Instead of asking why you think **const** is important, I want to start with a different question.

Why don't we use global variables?

Why Const?

- "Global variables can be read or modified by any part of the program, making it difficult to remember or reason about every possible use"
- "A global variable can be get or set by any part of the program, and any rules regarding its use can be easily broken or forgotten"

Why Const?

- "Non-const variables can be read or modified by any part of the function, making it difficult to remember or reason about every possible use"
- "A non-const variable can be get or set by any part of the function, and any rules regarding its use can be easily broken or forgotten"

Why Const?

Find the bug in this code:

```
void f(int x, int y) {  
    if ((x==2 && y==3) || (x==1))  
        cout << 'a' << endl;  
  
    if ((y==x-1)&&(x==-1 || y=-1))  
        cout << 'b' << endl;  
  
    if ((x==3)&&(y==2*x))  
        cout << 'c' << endl;  
}
```

Why Const?

Find the bug in this code:

```
void f(int x, int y) {  
    if ((x==2 && y==3) || (x==1))  
        cout << 'a' << endl;  
    if ((y==x-1)&&(x==-1 || y=-1))  
        cout << 'b' << endl;  
    if ((x==3)&&(y==2*x))  
        cout << 'c' << endl;  
}
```

Why Const?

Find the bug in this code:

```
void f(const int x, const int y) {  
    if ((x==2 && y==3) || (x==1))  
        cout << 'a' << endl;  
  
    if ((y==x-1)&&(x==-1 || y=-1))  
        cout << 'b' << endl;  
  
    if ((x==3)&&(y==2*x))  
        cout << 'c' << endl;  
}
```


Why Const?

The compiler finds the bug for us!

```
test.cpp: In function 'void f(int, int)':  
test.cpp:7:31: error: assignment of read-only parameter 'y'
```

Why Const?

That's a fairly basic use case though, is that really all that const is good for?

The const Model

Planet earth;



The const Model

```
long int countPeople(Planet& p);
```

```
long int population = countPeople(earth);
```

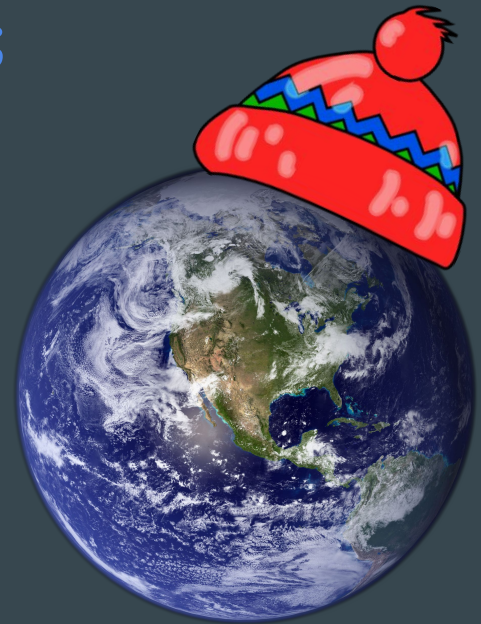


The const Model

```
addLittleHat(earth);
```



`countPeople(earth)`

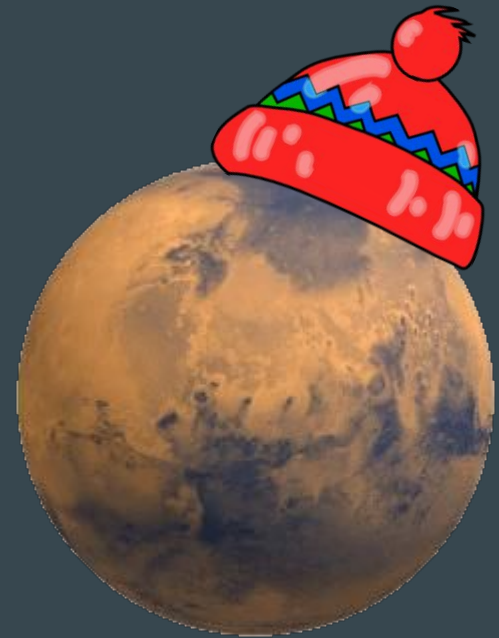


The const Model

```
marsify(earth);
```



countPeople(earth)

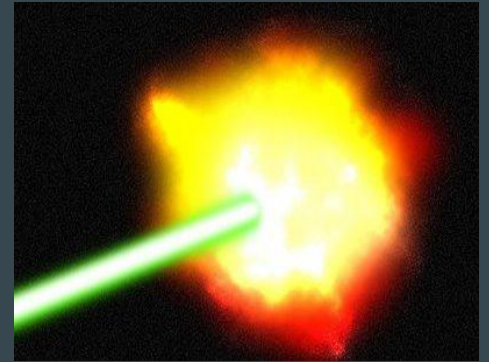


The const Model

```
deathStar(earth);
```



countPeople(earth)



Why Const?

How did this happen?

The const Model

```
long int countPopulation(Planet& p) {  
    // Hats are the cornerstone of modern society  
    addLittleHat(p);  
  
    // More land; oceans were wasting space  
    marsify(p);  
  
    // Optimization: destroy planet  
    // This makes population counting  $O(1)$   
    deathStar(p);  
    return 0;  
}
```

The const model

What would happen if I made that a const method?

The const Model

```
long int countPopulation(const Planet& p) {  
    // Hats are the cornerstone of modern society  
    addLittleHat(p);  
  
    // More land; oceans were wasting space  
    marsify(p);  
  
    // Optimization: destroy planet  
    // This makes population counting  $O(1)$   
    deathStar(p);  
    return 0;  
}
```

The const Model

test.cpp: In function 'long int countPopulation(const Planet&)':

test.cpp:9:21: error: invalid initialization of reference of type
'Planet&' from expression of type 'const Planet'

test.cpp:3:6: error: in passing argument 1 of 'void addLittleHat
(Planet&)'

test.cpp:12:12: error: invalid initialization of reference of type
'Planet&' from expression of type 'const Planet'

test.cpp:4:6: error: in passing argument 1 of 'void marsify(Planet&)'

test.cpp:16:14: error: invalid initialization of reference of type
'Planet&' from expression of type 'const Planet'

test.cpp:5:6: error: in passing argument 1 of 'void deathStar(Planet&)'

The const Model

const allows us to reason about whether a variable will be changed.

The const Model

```
void f(int& x) {  
    // The value of x here  
  
    aConstMethod(x);  
  
    anotherConstMethod(x);  
  
    // Is the same value of x here  
}
```

The const Model

```
void f(const int& x) {  
    // Whatever you want  
}  
  
void g() {  
    int x = 2;  
    f(x);  
    // x is still equal to two  
}
```

const and Classes

This is great for things like `ints`, but how does `const` interact with classes?

How do we define `const` member functions?

const and Classes



Let's have this cloud represent the member variables of a certain string

const and Classes



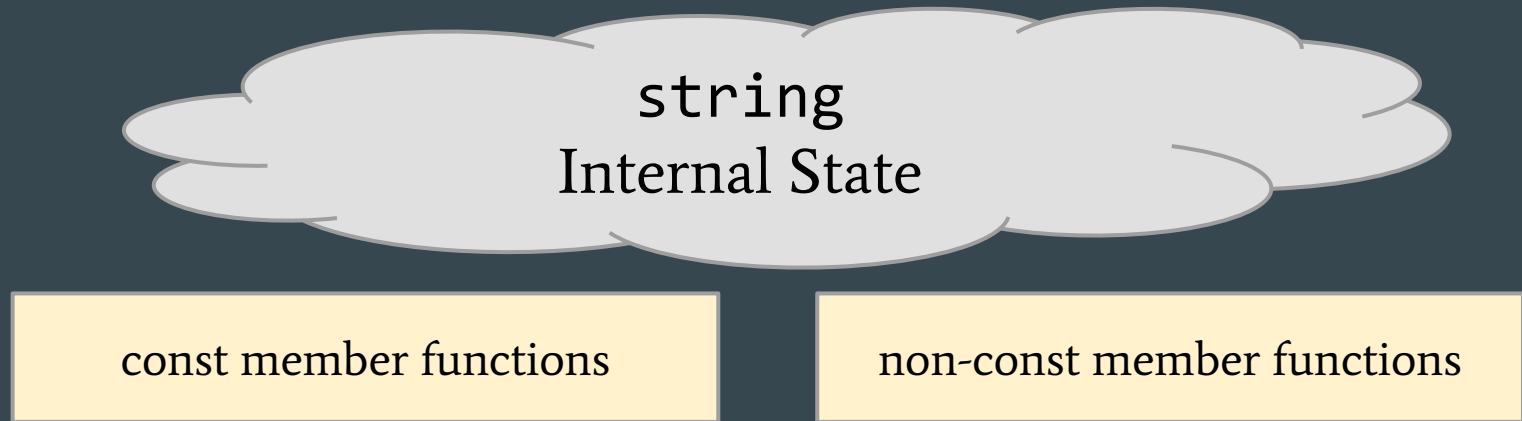
A diagram illustrating the components of a class. It features a light gray cloud-like shape at the top containing the text 'string' and 'Internal State'. Below this, a yellow rectangular box contains the text 'member functions'.

string
Internal State

member functions

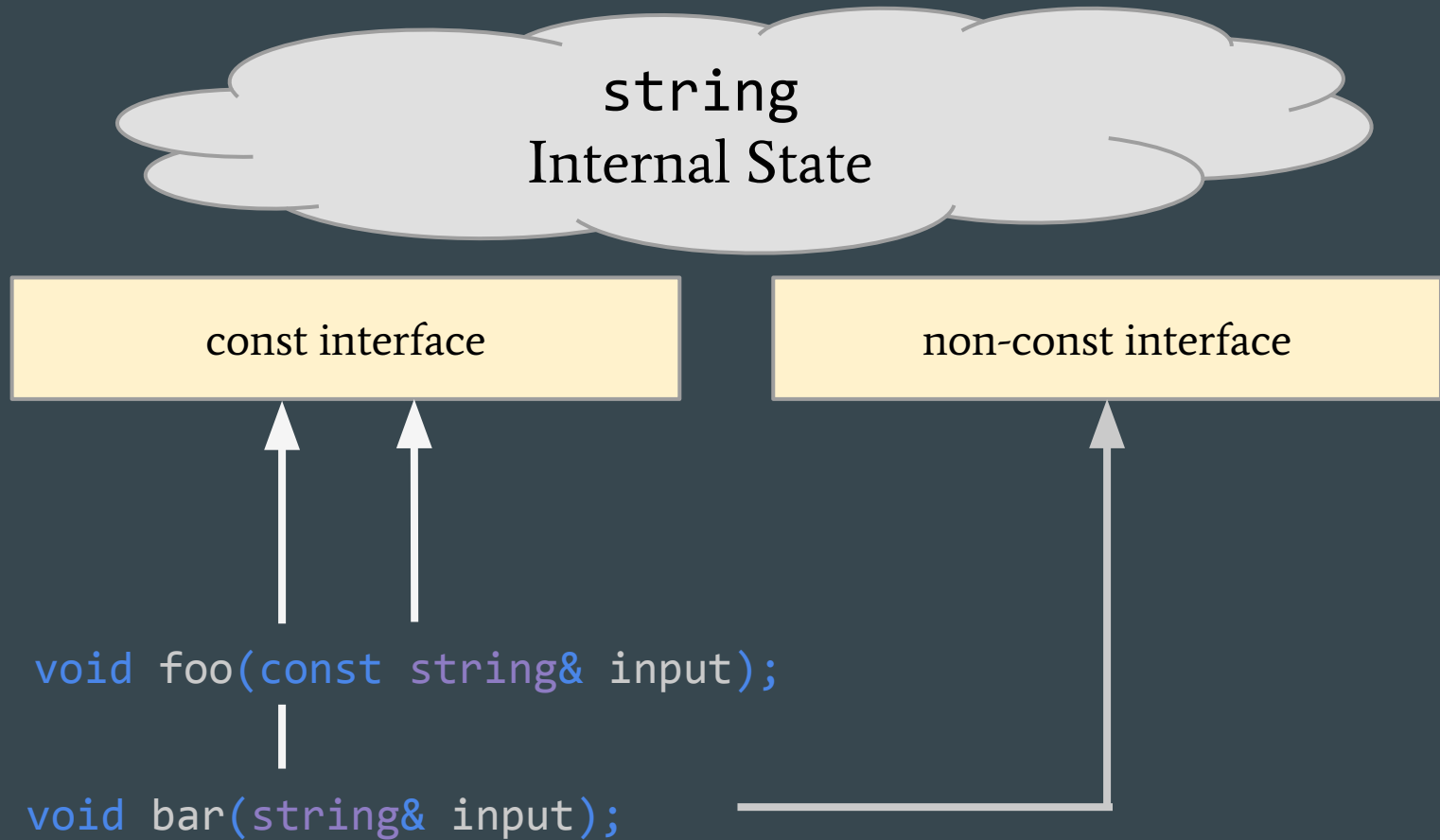
Previously, we thought that you just used member functions to interact with an instance of an object

const and Classes



Now we see that there are both const and non-const member functions,
and const objects can't use non-const member functions

const and Classes



The const Model

```
// Defining const member functions
struct Planet {
    int countPopulation() const;
    void deathStar();
};

int Planet::countPopulation() const {
    return 42; // seems about right
}

void Planet::deathStar() {
    cout << "BOOM" << endl;
}
```

The const Model

```
// using const member functions
struct Planet {
    int countPopulation() const;
    void deathStar();
};

void evil(const Planet &p) {
    // OK: countPopulation is const
    cout << p.countPopulation() << endl;
    // NOT OK: deathStar isn't const
    p.deathStar();
}
```

Adding Const to Vector

Let's go through as much of const as we can on vector

const and non-const Versions

Sometimes, you'll have two copies of a function, one const, one non-const:

```
const ElemType& Vector<ElemType>::at(size_t index) const {  
    return *(begin() + index);  
}
```

```
ElemType& Vector<ElemType>::at(size_t index){  
    return *(begin() + index);  
}
```

Can we avoid the duplicated code?

Removing Const

C++ provides a way to remove const called `const_cast`:

```
const_cast<Type>(/* a const value */);
```

Usage:

```
const int &constI = // some value  
  
int &i = const_cast<int&>(constI);
```

USE THIS ONLY WHEN NECESSARY

Issues with const Removal

- const is a safety measure. You are asking the compiler to ignore a blatant violation of this safety measure.
- const is a safety measure. You are asking the compiler to ignore a blatant violation of this safety measure.
- const is a safety measure. You are asking the compiler to ignore a blatant violation of this safety measure.
- const is a safety measure. You are asking the compiler to ignore a blatant violation of this safety measure.
- const is a safety measure. You are asking the compiler to ignore a blatant violation of this safety measure.

All of the above issues are important and worth considering.

Removing Duplicate Code: Option #1

```
const string& at(const Tree &tree, size_t index) {  
    // implementation  
}  
  
string& at(Tree &tree, size_t index) {  
    const Tree &constTree = tree;  
    return const_cast<string&>(at(constTree, index));  
}
```

Removing Duplicate Code: Option #2

```
const string& at(const Tree &tree, size_t index) {  
    Tree &nonConstTree = const_cast<Tree&>(tree);  
    return at(nonConstTree, index);  
}  
  
string& at(Tree &tree, size_t index) {  
    // implementation  
}
```

Which Option?

Always choose option #1 and have the non-const function call the const function.

Option #2 only works because, at the moment, the non-const function isn't actually doing anything non-const besides giving you a reference. Having the const function rely on this is dangerous, because, by definition, a non-const function doesn't have to obey that.

Removing Duplicate Code in Practice

```
template <typename ElemType>
const ElemType& Vector<ElemType>::at(size_t index) const {
    return *(begin() + index);
}
```

```
template <typename ElemType>
ElemType& Vector<ElemType>::at(size_t index) {
    // A bit weird at first, but it does what we want
    const Vector& constThis = *this;
    return const_cast<ElemType&>(constThis.at(index));
}
```

A Const Pointer

- Using pointers with const is a little tricky
 - When in doubt, read right to left

```
//constant pointer to a non-constant Widget  
Widget * const p;
```

```
//non-constant pointer to a constant Widget  
const Widget* p; Widget const* p;
```

```
//constant pointer to a constant Widget  
const Widget* const p;  
Widget const* const p;
```

Const Iterators

- Remember that iterators act like pointers
- `const vector<int>::iterator itr` however, acts like `int* const itr`
- To make an iterator read only, define a new `const_iterator`

Const Iterators

```
const vector<int>::iterator itr = v.begin();  
*itr = 5; //OK! changing what itr points to  
++itr; //BAD! can't modify itr
```

```
vector<int>::const_iterator itr = v.begin();  
*itr = 5; //BAD! can't change value of itr  
++itr; //OK! changing v  
int value = *itr; //OK! reading from itr
```

Recap

Where does const work?

It can be used as a **qualifier** on any type. This works for everything from arguments to local variables.

```
const string &s = f();
```

It can also be used on functions:

```
size_t Vector<ElemType>::size() const;
```

Recap

- For the most part, always anything that does not get modified should be marked `const`
- Pass by `const` reference is better than pass by value
 - Not true for primitives (`bool`, `int`, etc)
- Member functions should have both `const` and non `const` iterators
- Read right to left to understand pointers
- Please don't make a method to blow up earth

Final Notes

`const` on objects:

Guarantees that the object won't change by allowing you to call only `const` functions and treating all public members as if they were `const`. This helps the programmer write safe code, and also gives the compiler more information to use to optimize.

`const` on functions:

Guarantees that the function won't call anything but `const` functions, and won't modify any non-static, non-mutable members.

`nullptr`

Totally unrelated thing I should've mentioned forever ago!

`NULL` is old hat.

`nullptr` is the shiny new replacement.

Motivation

```
// Makes a time representing @seconds seconds from epoch
```

```
time_t time(int seconds);
```

```
// Same interface as in ctime.h
```

```
time_t time(time_t* timer);
```

```
time(NULL);    // Calls the second
```

```
time(nullptr); // Calls the second
```

Motivation

```
// Makes a time representing @seconds seconds from epoch
```

```
time_t time(int seconds);
```

```
// Same interface as in ctime.h
```

```
time_t time(time_t* timer);
```

```
time(NULL);    // Actually, it calls the first. Woops.
```

```
time(nullptr); // Calls the second
```

nullptr

Just like the NULL you know and love, but can't be implicitly converted to an integer!

NULL would rather act as an integral type than a pointer, nullptr is always a pointer.