Const Correctness

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Const Everything
- Prep for Next Time
Announcements
Recap
Let’s go back for a second...

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td>~</td>
<td>!</td>
<td>,</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;*</td>
<td>new</td>
<td>new</td>
<td>delete</td>
<td>delete</td>
</tr>
</tbody>
</table>
Non-member Functions

Add a function named `operator@` outside your class.

Have it take all its operands.

```cpp
bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```
Let's go back for a second...

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td>!</td>
<td>,</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
</tr>
</tbody>
</table>
### Operator Overloading

Let’s go back for a second...

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
<td>%</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td>~</td>
<td>!</td>
<td>,</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
<td>()</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-*</td>
<td>new</td>
<td>new []</td>
<td>delete</td>
<td>delete []</td>
<td></td>
</tr>
</tbody>
</table>

Anything curious here?
Let’s go back for a second...

### Operator Overloading

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td>~</td>
<td>!</td>
<td>,</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;*</td>
<td>new</td>
<td>new []</td>
<td>delete</td>
<td>delete []</td>
</tr>
</tbody>
</table>

Anything curious here?
Functors

Classes which define the () operator.

Why is this useful?

- Can have state
- Customizable through constructor

Very useful for algorithms!
Functors

Functors let us make customizable functions!

We can pass useful information to their constructor that was not known at compile time.

But...

Kind of a Pain™
Functors let us make customizable functions!

We can pass useful information to their constructor that was not known at compile time.

But...

Kind of a Pain™

C++11 has a solution!
Lambdas

A C++11 feature that lets you make functions on the fly.

```cpp
[capture-list](params) -> ReturnType {
    // code

};
```
Lambda Captures

A C++11 feature that lets you make functions on the fly.

\[
[capture-list](\text{params}) \rightarrow \text{ReturnType} \ 
\{
    // code
\};
\]

What is this for?
Lambda Captures

You can capture available variables to use in the lambda

[byValue, &byReference]

You can also capture all currently available variables:

[=] // By value

[&] // By reference

This will only capture the ones used inside the function.
How Lambdas Work?

```cpp
[capture-list](params) -> ReturnType {
    // code
};

class SomeName {
public:
    SomeName(capture-list) {
        // set each private member to
        // thing in capture list
    }

    ReturnType operator()(params) {
        // code
    }

private:
    // create private member for each
    // thing in capture-list
};
```
Complex Data Structures
std::queue

A First-In/First-Out (FiFO) data structure

```cpp
template<
    class T,
    class Container = std::deque<T>
>
class queue;
```
std::queue

Needs two template types to be constructed:

### Template parameters

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)
- **Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `front()`
  - `push_back()`
  - `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.
A priority queue lets us get the earliest event in the sequence!

```cpp
std::priority_queue
```

Defined in header `<queue>`

```cpp
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
>
class priority_queue;
```
std::priority_queue

Needs three template types to be constructed:

Template parameters

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)
- **Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer, and its iterators must satisfy the requirements of RandomAccessIterator. Additionally, it must provide the following functions with the usual semantics:
  - front()
  - push_back()
  - pop_back()

The standard containers std::vector and std::deque satisfy these requirements.

- **Compare** - A Compare type providing a strict weak ordering
How do we make a priority_queue of names ordered by last name?

```cpp
// Function to compare two names by last name
auto cmpFn = …;

// Create priority queue
std::priority_queue<
    vector<string>,
    vector<vector<string>>,
    decltype(cmpFn)> queue(cmpFn);
```
Const Correctness

Mike Precup (mprecup@stanford.edu)
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 \texttt{const} is important, I want to start with a different (related) 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:

```c
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;
}
```
Find the bug in this code:

```cpp
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?
Why Const?

Find the bug in this code:

```cpp
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?
Why Const?

No.
The const Model

Planet earth;
The const Model

```c
int countPeople(Planet& p);
//...
int population = countPeople(earth);
```
The const Model

addLittleHat(earth);

countPeople(earth)
The const Model

countPeople(earth);
marsify(earth);
The const Model

countPeople(earth);

defaultStar(earth);
Why Const?

How did this happen?
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?
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

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

```c
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?
Let's have this cloud represent the member variables of a certain string.
Previously, we thought that you just used member functions to interact with an instance of an object.
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

string
Internal State

const interface

void foo(const string& input);

non-const interface

void bar(string& input);
// 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();
}
A Const Pointer

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

```c
//constant pointer to a non-constant int
int * const p;  // (*p)++; OK!

// p++; NOT allowed!
```
A Const Pointer

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

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

//non-constant pointer to a constant int
const int* p;
```
A Const Pointer

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

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

//non-constant pointer to a constant int
const int* p;
int const* p;
```
A Const Pointer

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

//constant pointer to a non-constant int
int * const p;

//non-constant pointer to a constant int
const int* p;
int const* p;

//constant pointer to a constant int
const int* const p;
int const* const p;
A Const Pointer

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

```c
//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`

```cpp
vector v{1,2312};

const vector<int>::iterator itr = v.begin();
++itr;  // doesnt compile

*itr = 15;  // compiles
```
Const Iterators

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

```cpp
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.

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

It can also be used on functions:

```cpp
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.