Kevin Miller
CS 106B

Handout #51
March 6, 2015

The STL and Const Correctness

The STL & const correctness

Kevin Miller
(kmiller4@stanford.edu)

Roadmap

- Today we will bridge the gap between the Stanford Libraries and the Standard Template Libraries (STL) used in the real world
- We will look at
  - STL containers
  - const correctness

Review: Sequence Containers

- A container class allows you to store any number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (and many more!).

What I Want To Show You

- Why the Stanford library exists
- How to use STL sequence containers instead of the Stanford Library
  - We'll look at the differences between STL/Stanford using stack and vector, and we'll also examine a new STL class, deque
- Performance of different containers, and why you might choose one over another

Why the Stanford Library Exists

Students often ask:

"Why do we need to use the Stanford libraries in CS106B/X?"

Why the Stanford Library Exists

- The Stanford libraries include things not found in the STL (`Grid`, `getInteger` and friends, graphics).
- Many parts of the Stanford library give up performance for simplicity
- Debugging Stanford library code can be much easier than debugging STL code

Container #1: Stack

First, let's talk about how to use the STL stack.

STL <stack>: What's Similar

What you want to do	Stanford Stack<int>	STL stack<int>
Create a stack	`Stack<int> x;`	`stack<int> x;`
Get the size of a stack	`int size = x.size();`	`int size = x.size();`
Check if a stack is empty	`if (x.isEmpty()) ...`	`if (x.empty()) ...`
Push a value on the stack	`x.push(42);`	`x.push(42);`
Peek at the top element without popping it	`int top = x.peek();`	`int top = x.top();`
Pop off the top element and ignore its value	`x.pop();`	`x.pop();`

STL <stack>: What's Different

What you want to do	Stanford Stack<int>	STL stack<int>
Clear the stack	`x.clear();`	`while(!x.empty())` `  x.pop();`

Convert the stack to a string	`string s =`
	`x.toString();`
		`string s;`
		`while(!x.empty() {`
		`  s += x.top();`
		`  s += " ";`
		`  x.pop();`
		`}`

Pop and save the value	`int top = x.pop();`	`int top = x.top();` `x.pop();`

STL <stack>: Why the differences?

Looking at the differences between the STL and the Stanford libraries can help you understand the the reason each of these libraries were designed.

"Thus, the standard library will serve as both a tool and as a teacher"
- Bjarne Stroustrup

STL <stack>: Why the differences?

Why is there no .clear() function for stacks?

- Conceptually, clearing isn't part of the interface to a stack
- It's very easy to write your own clear function:

```
// stack<int> s = ...;
while (!s.empty()) {
  s.pop();
}
```

STL <stack>: Why the differences?

Why doesn't pop return the value it removed?

- The caller might not need the value, in which case returning the value would be wasteful.
- It's easy to write code which pops and saves the value.

```
// stack<int> s = ...;
int value = s.top();
s.pop();
```

STL <stack>: Why the differences?

Why isn't there a `toString` function?

- Implementing toString would require that the type stored in the stack could be converted to a string
  - For example, you can convert a `stack<int>` to a `string` because you can convert an `int` to a `string`.
- It's tough to say what the "proper" way to convert a stack to a string is

Container #2: Vector

Next, let's talk about how vectors are represented in the STL.

STL <vector>: What's Similar

What you want to do	Stanford Vector<int>	STL vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Add a value k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear a vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`

STL <vector>: What's Different

Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`
Apply a function to each element in x	`x.mapAll(fn)`	`// Impossible! (come back on Wednesday)`
Concatenate vectors v1 and v2	`v1 += v2;`	`// Impossible! (come back on Wednesday)`
Add an element to the beginning of a vector	`// Impossible! (or at least slow)`	`// Impossible! (or at least slow)`

STL <vector>: Why the differences?

Why doesn't vector have bounds checking?

- If you write your program correctly, bounds checking will do nothing but make your code run slower

STL <vector>: Why the differences?

Why is there no push_front method?

- This is a bit more complicated

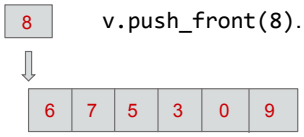
The Mystery of push_front

Pushing an element to the front of the vector requires shifting all other elements in the vector down by one, which can be **very** slow

To demonstrate this, let's say we had this nice little vector:

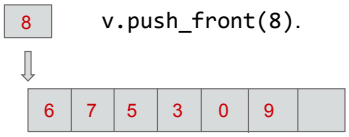
The Mystery of push_front

Now, let's say that push_front existed, and that you wanted to insert an 8 at the beginning of this vector.



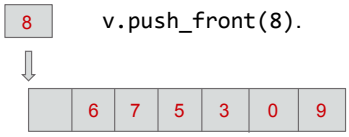
The Mystery of push_front

First, we may have to expand the capacity of the vector



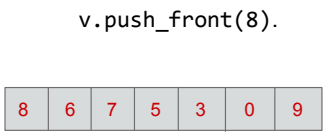
The Mystery of push_front

Then, we'll need to shift every single element down one position



The Mystery of push_front

Finally, we can actually insert the element we wanted to insert.



Just how bad is push_front?

```
// Adding to the back
for (int i = 0; i < N; i++)
     v.push_back(i);

// Or: Adding to the front
for (int i = 0; i < N; i++)
     v.insert(v.begin(), i);

// How big can the difference be?
```

Just how bad is push_front?

	push_front	push_back
N = 1000	0.01	0
N = 10000	0.89	0.01
N = 100000	117.98	0.04
N = 1000000	Hours	0.31
N = 10000000	Years	3.16

You can see the difference between an $O(n^2)$ algorithm and an O(n) algorithm!

STL <deque>: What's a deque?

- A deque (pronounced "deck") is a **d**ouble **e**nded **que**ue
- Unlike a vector, it's possible (and fast) to push_front
- The implementation of a deque isn't as straightforward as a vector though

STL <deque>: Performance

- We can now use the `push_front` function, and it will run much faster than if we had used a vector.
- Let's see how this looks in real world performance numbers

push_front: vector and deque

```
// Vector test code
vector<int> v;
// Insert at the start of the vector
for (int i = 0; i < N; i++)
     v.insert(v.begin(), i);
// Clear by using pop_front (erase)
for (int i = 0; i < N; i++)
     v.erase(v.begin());
```

push_front: vector and deque

```
// Deque test code
deque<int> d;
// Insert elements using push_front
for (int i = 0; i < N; i++)
     d.push_front(i);
// Clear by using pop_front
for (int i = 0; i < N; i++)
     d.pop_front();
```

push_front: vector and deque

	<vector>	<deque>
N = 1000	0.02	0
N = 10000	2.12	0.01
N = 100000	264.9	0.04
N = 1000000	Years	0.44
N = 10000000	Millenia	5.54

Why use a vector?

If a deque can do everything a vector can plus add to the beginning, why not always user deques?

- For other common operations like access and adding to the end, a vector outperforms a deque

Element Access: vector and deque

```
vector<int> v(N);
deque<int> d(N);

for (int i = 0; i < N; i++)
     v[i] = i;

for (int i = 0; i < N; i++)
     d[i] = i;
```

Access: vector and deque

	<vector>	<deque>
N = 1000	0.02	0.14
N = 10000	0.28	1.32
N = 100000	3.02	13.22
N = 1000000	30.84	133.30

Other Sequence Containers

The STL also includes priority queue, queue, and linked list classes, but those aren't too important to us right now.

Associative Containers

- Like Sequence Containers, Associative containers store data
- Unlike Sequence Containers, Associative containers have no idea of an ordering
- Instead, based on a **key**
- We will look at Map and Set

STL <map>

- Methods are the same as the Stanford Map except for some syntax differences
  - If you want to see a complete list of methods, google search std::map or check out http://www.cplusplus.com/reference/map/map
- Works the exact same way as Stanford Map when using [ ]

STL <set>

- Methods are the same as the Stanford Set except for some syntax differences
  - If you want to see a complete list of methods, google search std::set or check out http://www.cplusplus.com/reference/set/set/
- Key point, a set is just a specific case of a map

Const

Enough about containers, let's talk about const

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;
```

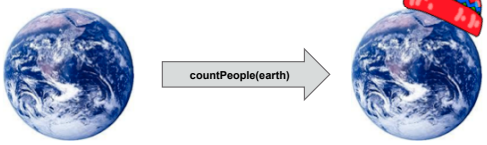
The const Model

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

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

The const Model

```
addLittleHat(earth);
```

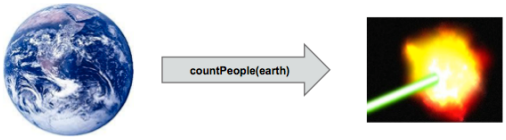


The const Model

```
marsify(earth);
```

The const Model

```
deathStar(earth);
```



Why Const?

How did this happen?

The const Model

```
long int countPopulation(Planet& p) {
      // I don't like people not wearing hats
  addLittleHat(p);

  // Mars-like planets are easier to deal with
  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) {
      // I don't like people not wearing hats
  addLittleHat(p);

  // Mars-like planets are easier to deal with
  marsify(p);

  // Optimization: destroy planet
  // This makes people 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) {
    // Anything whatsoever
}
void g() {
  int x = 2;
  f(x);
  // x is still equal to two
}
```

const and Classes

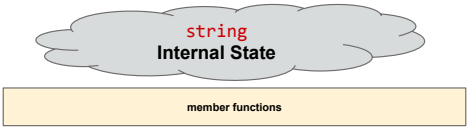
This is great for things like `int`s, but how does `const` interact with classes?

How do we define `const` member functions?

const and Classes

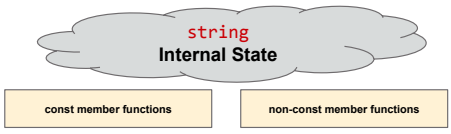
Let's have this cloud represent the member variables of a certain string

const and Classes



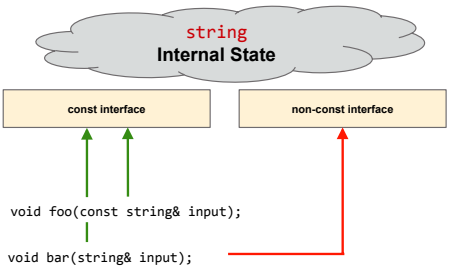
Previously, we thought that you just used member functions to interact with an instance of an object

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 Plant::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

```
//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;
```

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
- 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