Const Correctness

CS 106L Spring 2020 – Avery Wang and Anna Zeng
Stanford University

11 May 2020

Key questions we will answer today

- try a fairly complex example of template classes
- introduce const correctness
- (if time permits) introduce const_iterator

Where we're going...

CS 106B has covered the absolute barebones of C++ class design.

We will cover the rest:

- template classes
- **const correctness**
- operator overloading
- special member functions
- move semantics
- RAII

Game Plan

- const reference
- const members
- const_iterator

review: const references

Thinking about const pointers.

- A reference is an alias to a variable.
- A const reference allows you to read that variable through the alias, but not modify that variable.

```
int main() {
    int i = 3;
    const int& ref = i;
    i++;
    cout << ref << endl;  // okay, prints 4
    ref = 1;              // error, const
}
```

We've seen this chart before, and we'll finish the last two rows next week!

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

const member functions

Today's Goal: make sense of this problem

```
// main.cpp
#include "vector.h"

void foo(const vector<int>& vec) {
   cout << vec.size() << endl;
}
```

Today's Goal: make sense of this problem

```cpp
// vector.h
template <typename T>
class vector {
   size_t size();
}

template <typename T>
size_t vector<T>::size(){

}
```

```cpp
// main.cpp
#include "vector.h"

void foo(const vector<int>& vec) {
   cout << vec.size() << endl;
}
```

```
main.cpp:10:13: error: 'this' argument to member function 'size' has type 'const vector<int>', but function is not marked const
    cout << vec.size() << endl;
            ^~~
./vector.h:30:16: note: 'size' declared here
       size_t size();
              ^
1 warning and 1 error generated.
make: *** [main] Error 1
```

Const Members

- A const member function is a function that cannot modify any private member variables.
- A const member function can only call other const member functions.

Const Members

- Why should we enforce const correctness?
- Const correctness is not just a style thing - it's necessary for your code to be correct!

Const-qualified object

- A const-qualified object (instance) is one that cannot be modified (after its constructor).
- The object may be constructed as const (rare), or you might have a const reference to a non-const object (common).

Const-qualified object

- Since a const-qualified object cannot modify its members, it can only call its own const member functions
- Exception: constructor and destructor

Summary of Terminology

- const reference = a reference that cannot be used to modify the object that is being referenced.
- const method = a method of a class that can't change any class variables and can only call other const methods.
- const object = an object declared as const that can only call its const methods.

Today's Goal: make sense of this problem

```cpp
// vector.h
template <typename T>
class vector {
   size_t size() const;
}

template <typename T>
size_t vector<T>::size() const {

}
```

```cpp
// main.cpp
#include "vector.h"

void foo(const vector<int>& vec) {
   cout << vec.size() << endl;
}
```

```
main.cpp:10:13: error: 'this' argument to member function 'size' has type 'const vector<int>', but function is not marked const
    cout << vec.size() << endl;
            ^~~
./vector.h:30:16: note: 'size' declared here
       size_t size();
              ^
1 warning and 1 error generated.
make: *** [main] Error 1
```

Original function from last week.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Original function from last week.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Problem: in the const vector we should be able to still call at.

Attempt 1: make at() a const function.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Attempt 1: make at() a const function.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Problem: at returns a non-const reference, allowing you to modify values inside the vector!

Attempt 2: have at() return non-const reference.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   const T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   const T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Attempt 2: have at() return non-const reference.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   const T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   const T& at(size_t index) const;

   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Problem: non-const vector needs to return a non-const reference.

Attempt 3: include both const and non-const.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);
   const T& at(size_t index) const;
   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);
   const T& at(size_t index) const;
   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Attempt 3: include both const and non-const.

```
// non-const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);
   const T& at(size_t index) const;
   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

```
// const objects
template <typename T>
class vector {
   vector();
   ~vector();
   T& at(size_t index);
   const T& at(size_t index) const;
   void push_back(const T& elem);
   size_t size() const;
   bool empty() const;
   // among other functions
}
```

Compiler will prefer the non-const version if both versions are possible.

Implementation for both look the same.

```
template <typename T>
T&
vector<T>::at(size_t index) {
   return _elems[index];
}
```

```
template <typename T>
const T&
vector<T>::at(size_t index) const {
   return _elems[index];
}
```

There's not too much code, so code duplication is fine…
ask us if you want to know a better way!

const_iterators

const_iterator != const iterator

- iterator points to non-const objects
- const_iterator points to const objects

- The const_iterator object itself is not const!
- You can perform ++ on a const_iterator.
- You cannot dereference and write to a const_iterator (*iter = 3)

The type of iterator provided depends on the const-ness of the container.

- Non-const containers provide iterators
- const containers provide const_iterators (since their internal elements are const)

- Makes intuitive sense: const_iterators don't let you write to the const containers.

A const iterator is a const object (i.e. can't be changed)

- A const iterator cannot be changed after it is constructed.
- No incrementing or reassignment is allowed.
- You CAN dereference a const iterator and write to it!

template classes exercise

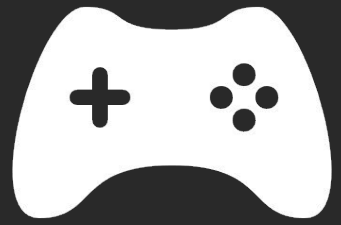
Interesting Template Exercise

- Implement a data structure which allows you to insert elements one at a time, and query the median of all elements added so far at any time.

- Create this class with your group of 3-4 students, and we'll regroup at 4:45.
- There are 4 short methods to implement.
- Aim for simplicity!
- If time remains, try to make your solution as efficient as possible.

Key Takeways from the Exercise

- You can have multiple class template parameters.
- You can have a default parameter values and default template types.
- You can design template classes to accept types that are functions, such as comparison functions.
- decltype from A1 is back!
- Lots of review of template syntax.
- Lots of review of algorithms and lambdas!
- A2 is extending a template class that has a template function parameter "hash function".



Next time

Operator Overloading

More to learn!

Note: we got feedback that the supplemental slides were a bit confusing, so we will remove them.

None of these are important for the remainder of the class (including the assignment). Come chat with us if you want to learn more.

- const pointers (const int* vs. int const * vs. int * const)
- static_cast/const_cast trick (design pattern on how to unify the code between const and non-const versions).
- constexpr (C++11), constval (C++20), constinit (C++20)