Lecture 11: Const correctness

CS106L, Winter '21
CS 106B covers the **barebones** of C++ classes

we’ll be covering the rest

template classes • const correctness • operator overloading
special member functions • move semantics • RAII
CS 106B covers the **barebones** of C++ classes

we’ll be covering the rest

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

- Recap: template classes
- Recap: const references
- Const members
- const_iterator
Quick recap: template class code from last time
Recap: const references
Const references

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

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

What's going to happen if we compile this code? (answer in chat)
Today’s goal: make sense of this problem

What’s going to happen if we compile this code? (answer in chat)
🤔 Questions? 🤔
Understanding const correctness
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
If a function should be able to be called on a const object, it should be designated as const.

Why should we enforce const correctness?

- Const correctness isn’t just a style thing - it’s necessary for your code to be correct! (example: see compiler error a few slides before this)
Const-qualified object

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

```cpp
const std::vector<int> my_vec{1, 2, 3}; // Object declared as const
std::vector<int> other_vec;
const std::vector<int>& ref = other_vec; // Const reference

std::cout << my_vec.size() << endl; // allowed
my_vec.push_back(4); // not allowed!
```
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.
So, how do we fix this code?

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

#include vector.cpp

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

void foo(const vector<int>& vec) {
    cout << vec.size() << endl;
}
Just add a const keyword!

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

#include vector.cpp

// vector.cpp
template <typename T>
size_t vector<T>::size() const {
    // some code
}

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

void foo(const vector<int>& vec) {
    cout << vec.size() << endl;
}
```
Const objects see a subset of member functions

What non-const vectors see

// 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
}

What const vectors see

// 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
}
Do you see any potential problems here? (chat)

What non-const vectors see

```
// 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
}
```

What const vectors see

```
// 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
}
```
Ans: in the const vector we should still be able to call .at()!

What non-const vectors see

// 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
}

What const vectors see

// 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
}
**Attempt 1: Make at() a const function**

**What non-const vectors see**

```cpp
// 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
}
```

**What const vectors see**

```cpp
// 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!

**What non-const vectors see**

```cpp
// 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
}
```

**What const vectors see**

```cpp
// 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 2: Have at() return a const reference

What non-const vectors see

// 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
}

What const vectors see

// 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!

What non-const vectors see

```cpp
// 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
}
```

What const vectors see

```cpp
// 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 3: include both const and non-const

What non-const vectors see

```cpp
// non-const objects
template <typename T>
class vector {
    vector();
    ~vector();

    const T& at(size_t index) const;
    T& at(size_t index);
    void push_back(const T& elem);

    size_t size() const;
    bool empty() const;
    // among other functions
}
```

What const vectors see

```cpp
// const objects
template <typename T>
class vector {
    vector();
    ~vector();

    const T& at(size_t index) const;
    T& at(size_t index);
    void push_back(const T& elem);

    size_t size() const;
    bool empty() const;
    // among other functions
}
```
Attempt 3: include both const and non-const

What non-const vectors see

// non-const objects
template <typename T>
class vector {
    vector();
    ~vector();

    const T& at(size_t index) const;
    T& at(size_t index);
    void push_back(const T& elem);

    size_t size() const;
    bool empty() const;
    // among other functions
}

What const vectors see

// const objects
template <typename T>
class vector {
    vector();
    ~vector();

    const T& at(size_t index) const;
    T& at(size_t index);
    void push_back(const T& elem);

    size_t size() const;
    bool empty() const;
    // among other functions
}

Note: compiler will prefer the non-const version if it’s not clear which one is being invoked
Implementation for both look the same!

Template implementations:

```
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 in this situation.
🤔 Questions? 🤔
Live Code Demo:
Let’s see a const-correct version of our vector class!
Announcements
Announcements

● Assignment 1 is due on Sunday at 11:59 PM
  ○ You can take up to two late days (until Tuesday at 11:59 PM)
● There might be some variance in the Emu->Stanford ladder. Don’t worry too much about this!
● Panopto is enabled!
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 write to a const_iterator (*iter = 3)

```cpp
std::vector<int> non_const_vec{1, 2, 3};
const std::vector<int> const_vec{1, 2, 3};

auto iter = non_const_vec.begin(); // non-const iter
const auto iter2 = non_const_vec.begin(); // const iter
auto iter3 = const_vec.begin(); // const_iterator
```
Type of iterator depends on const-ness of 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 (can't be changed)

- A const iterator cannot be changed after it is constructed.
- No incrementing or reassignment is allowed.

```cpp
std::vector<int> non_const_vec{1, 2, 3};
const auto iter2 = non_const_vec.begin(); // const iter
```
How const_iterators are used

https://en.cppreference.com/w/cpp/container/vector/begin
🤔 Questions? 🤔
Next time

Operator Overloading