Iterators and `const`

Remember `IntList`?

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();     // etc...
  };
  iterator begin() const;
  iterator end() const;
  // etc...
};
```

Using the Iterator

We shouldn't be allowed to change the contents!

```
void someFunction(const IntList& lst) {
   // Print every item
   for (IntList::iterator i = lst.begin();
            i != lst.end(); ++i) {
      cout << *i << endl;
   }
   // Change every item to 3
   for (IntList::iterator i = lst.begin();
            i != lst.end(); ++i) {
      *i = 3;
   }
}
```

The Problem

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();     // etc...
  };
  iterator begin() const;
  iterator end() const;
  // etc...
};
```

The Problem

```
class IntList {
 public:
  class iterator {
const ?int& operator*() const;
    iterator& operator++();     // etc...
  };But then we can't *ever* use an iterator to change contents!
  iterator begin() const;?
                              But then we can't get an iterator to a const list!
  iterator end() const;
  // etc...
};
```

~~Switch to Java?~~

Summary

- The iterators we've implemented have a problem
  - You can use them to change contents of the container
  - Even if the container is const!
- We need an iterator that lets you see but not change items

Coming Up…

Iterators that let you see but not change contents const_iterator

Big Idea

- An `iterator` lets you iterate through the structure
  - You can see and change the item at the iterator’s location
- A `const_iterator` lets you iterate through the structure
  - You can see but **not** change the item at the iterator’s location
- Not to be confused with a `const iterator`…
  - An `iterator` that is `const`
  - You can see and change the item at the iterator’s location
  - But you can’t change the iterator’s location

Return to `IntList`

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();        // etc...
  };
  class const_iterator {
    const int& operator*() const;
    const_iterator& operator++();  // etc...
  };

  iterator begin();
  iterator end();

  const_iterator cbegin() const;
  const_iterator cend() const;
  // etc...
```

Return to `IntList`

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();        // etc...
  };
  class const_iterator {
    const int& operator*() const;
    const_iterator& operator++();  // etc...
  };

  iterator begin();
  iterator end();

  const_iterator begin() const;
  const_iterator end() const;

  const_iterator cbegin() const;
  const_iterator cend() const;
  // etc...
```

When begin is called on a non-const list, an iterator is returned

When begin is called on a const list, a const_iterator is returned

cbegin always returns a const_iterator

Summary

- A `const_iterator` lets you see but not change items
- `cbegin` returns a `const_iterator`
- Usually we have two versions of begin:
  - The non-`const` version returns an `iterator`
  - The `const` version returns a `const_iterator`
  - Which version gets called depends on whether the collection is `const`

Coming Up…

Find the compiler errors!

Iterator Examples

FIND THE
COMPILER ERRORS!

```
void lstTest(IntList& lst) {
✓ IntList::iterator it = lst.begin();
✓ ++it;
✓ int x = *it;
✓ *it = 5;
};
```

```
void lstTest(IntList& lst) {
IntList::const_iterator it = lst.cbegin();
++it;
int x = *it;
*it = 5;
};
```

Can’t change the contents!

```
void lstTest(IntList& lst) {
const IntList::iterator it = lst.begin();
++it;
int x = *it;
*it = 5;
};
```

Can't move the iterator!

```
void lstTest(IntList& lst) {
const IntList::const_iterator it = lst.cbegin();
++it;
int x = *it;
*it = 5;
};
```

Can't move the iterator!

Can't change the contents!

```
void lstTest(IntList& lst) {
✗ IntList::const_iterator it = lst.begin();
✓ ++it;
✓ int x = *it;
✗ *it = 5;
};
```

begin returns an iterator not a const_iterator

Can't change the contents!

```
void clstTest(const IntList& lst) {
✓ IntList::const_iterator it = lst.begin();
✓ ++it;
✓ int x = *it;
✗ *it = 5;
};
```

Can't change the contents!

```
void clstTest(const IntList& lst) {
IntList::iterator it = lst.begin();
++it;
int x = *it;
*it = 5;
};
```

begin returns a const_iterator not an iterator!

```
void clstTest(const IntList& lst) {
✓ IntList::const_iterator it = lst.cbegin();
✓ ++it;
✓ int x = *it;
✗ *it = 5;
};
```

Can't change the contents!

YOU WIN!

Summary

- An `iterator` can move and change contents
- A `const_iterator` can move but not change contents
- A `const iterator` can change contents but not move
- A `const const_iterator` can neither move nor change contents
- When the collection is `const`, `begin` returns a `const_iterator`
- When the collection is non-`const`, `begin` returns an `iterator`
- Either way `cbegin` returns a `const_iterator`

Coming Up…

Implementing `const_iterators`

Implementing `const_iterators`

The Issue

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();       // etc...
  };
  class const_iterator {
    const int& operator*() const;
    const_iterator& operator++(); // etc...
  };

  iterator begin();
  iterator end();

  const_iterator begin() const;
  const_iterator end() const;

  const_iterator cbegin() const;
  const_iterator cend() const;
  // etc...
```

Code duplication!

Template Shenanigans

int if useInt is true
double if useInt is false

Conditional...type!

```
#include <type_traits>

bool useInt = true;

using val_t = std::conditional<useInt, int, double>::type;

val_t x;
```

x is either an int or double, depending on useInt

Toward a Solution

```
class IntList {
 public:
  class iterator {
    int& operator*() const;
    iterator& operator++();       // etc...
  };
  class const_iterator {
    const int& operator*() const;
    const_iterator& operator++(); // etc...
  };

  iterator begin();
  iterator end();

  const_iterator begin() const;
  const_iterator end() const;

  const_iterator cbegin() const;
  const_iterator cend() const;
  // etc...
```

Toward a Solution

```
class IntList {
 public:
  class iterator {
   public:
    using value_type = int;
    using reference = value_type&;
    using pointer = value_type*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    reference operator*() const;
    iterator& operator++();
    // etc...
  };
```

Toward a Solution

```
class IntList {
 private:
  class Iterator {
   public:
    using value_type = int;
    using reference = value_type&;
    using pointer = value_type*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    reference operator*() const;
    Iterator& operator++();
    // etc...
  };
 public:
  using iterator = Iterator;
```

Sometimes we want this to be a const reference.
Sometimes we don't...

Toward a Solution

```
class IntList {
 private:
  template <bool isConst>
  class Iterator {
   public:
    using value_type = int;
    using reference = value_type&;
    using pointer = value_type*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    reference operator*() const;
    Iterator<isConst>& operator++();
    // etc...
  };
 public:
  using iterator = Iterator<false>;
  using const_iterator = Iterator<true>;
```

Iterator is not a class anymore!
We have to give a template parameter.

Toward a Solution

```
class IntList {
 private:
  template <bool isConst>
  class Iterator {
   public:
    using value_type = int;
    using reference = std::conditional<isConst, const value_type&, value_type&>::type;
    using pointer = std::conditional<isConst, const value_type*, value_type*>::type;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    reference operator*() const;
    Iterator<isConst>& operator++();
    // etc...
  };
 public:
  using iterator = Iterator<false>;
  using const_iterator = Iterator<true>;
```

The rest should just work!

(use iterator and const_iterator as usual)

We've Arrived! The C++ Idiom:

```
class IntList {
 private:
  template <bool isConst>
  class Iterator {
   public:
    using value_type = int;
    using reference = std::conditional<isConst, const value_type&, value_type&>::type;
    using pointer = std::conditional<isConst, const value_type*, value_type*>::type;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    reference operator*() const;
    Iterator<isConst>& operator++();
    // etc...
  };
 public:
  using iterator = Iterator<false>;
  using const_iterator = Iterator<true>;
```

The rest should just work!
(use iterator and const_iterator as usual)

Summary

- A const_iterator could just be copy-pasted from an iterator
- There is a clever, complicated way around that
  - Give the iterator class a template variable to determine whether it should be const
  - Use a conditional type to set the reference type to const reference or not
- This is a reliable, idiomatic approach
  - You can apply it to your iterators!

Coming Up…

One awesome trick that you won't want to miss!

Better Living Through the `auto` Keyword

Types are Complicated

Now you can have variables of type…

```
std::vector<std::map<std::string, std::vector<st
```

Types are Complicated

Now you can have variables of type…

```
std::vector<std::map<std::string, std::vector<std::string::iterator> >::cons
```

Types are Complicated

Now you can have variables of type…

```
std::vector<std::map<std::string, std::vector<std::string::iterator> >::const_iterator x;
```

C++ “Classic”

```
void printNames(const std::list<std::string>& lst) {
  for (std::list<std::string>::const_iterator
         i = lst.cbegin(); i != lst.cend(); ++i) {
    std::cout << *i << std::endl;
  }
}
```

auto (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (auto i = lst.cbegin(); i != lst.cend(); ++i) {
    std::cout << *i << std::endl;
  }
}
```

auto means: “The same type as the value being assigned”

auto All The Things!

```
auto f(Row& r) {
  auto i = r.begin();
  auto x = *i;
  auto a = *x;
  auto e = doTheMagic(a);
  auto n = makeTheThing();
  return e*n;
}
```

`auto` ~~All The~~ Some of the Things!

- Use `auto` when it makes your code more readable
  - When the type genuinely doesn’t matter
  - When the type is clear from context
  - When the type is super complicated
- Avoid it when it hurts readability
  - When the type is simple and known
  - When the type is important for understanding the code
  - When the type is unclear from context

Summary

- `auto` means “the same type as the thing being assigned”
- It’s useful when the type is complicated or irrelevant
- Careful not to overuse it!

Coming Up…

Another quality of life improvement

Range `for` Loops
for Fun and Profit

C++ “Classic”

```
void printNames(const std::list<std::string>& lst) {
  for (std::list<std::string>::const_iterator
         i = lst.cbegin(); i != lst.cend(); ++i) {
    std::cout << *i << std::endl;
  }
}
```

Range for (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (std::string x : lst) {
    std::cout << x << std::endl;
  }
}
```

Use lst's iterator
Assign each item to x in turn

Similar to Python!
for x in lst:
  print(x)

Range for (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (const std::string x& : lst) {
    std::cout << x << std::endl;
  }
}
```

Prevent unnecessary copies

Range for (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (auto x : lst) {
    std::cout << x << std::endl;
  }
}
```

Is x a copy or a reference??

Common Pitfall: auto and References

```
int a = 5;

int& b = a;
```
b is another name for a!

```
auto c = b;
```
So c is an int (a copy of a)

```
auto& d = b;
```
d is a reference to a

Range for (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (auto x : lst) {
    std::cout << x << std::endl;
  }
}
```

Is x a copy or a reference?? Copy!!

Range for (Since C++11)

```
void printNames(const std::list<std::string>& lst) {
  for (const auto& x : lst) {
    std::cout << x << std::endl;
  }
}
```

Summary

- Range `for` loops simplify `for` loops over collections
  - Python-like syntax!
- Use references to avoid making copies!
- Also: be careful about `auto` and references!