# Note about naming in lecture slides

Today, all instances of "vector" refer to the vector we wrote in namespace mycollection, not std::vector.

We will either refer to this as mycollection::vector, or simply put using namespace mycollection above without using namespace std.

# Warmup!

- Walk through the .h and .cpp files of our vector class.
- Collectively identify:
  - things you've seen before and you can reason why it makes sense in this context
  - things you've seen before but you aren't sure why it makes sense in this context
  - things you haven't seen before
- We'll regroup in 10 minutes.

- If your group finishes earlier and is really bored, here's a challenge: the code seems like it works fine, but there's one instance of undefined behavior. Can you find/fix it?

# Template Classes

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

6 May 2020

# Key questions we will answer today

- what is a template class and how do we create them?
- where do template classes appear in the STL?
- what exactly is the :: (scope resolution) operator?
- what's the deal with .h vs. .cpp files?

# 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

# Thanks for the Feedback!

- Too fast! Too slow! Too many concepts in class. Please go faster! (we're trying to find a balance)
- More complicated code examples. (will do outside of class)
- Lectures too long. (we'll make them shorter)
- Tell us which supplemental material are important for the assignments. (none of them!)
- More centralized resources. (we'll make a central Piazza post)
- "occurrences" has been spelled correctly once on the slides (I didn't know that - please let us know if you find typos!)

# Game Plan

- file organization
- template aliases
- member templates

# Review of Classes

# Review: Classes

- A class defines a user-defined type (like a struct).
- By default, all members are private (unlike a struct).
- Classes give you more fancy C++ features than a struct.

# review of class design

- **interface**: specifies the operations that can be performed on instances of the class.

- **implementation**: specifying how those operations are to be performed.

# class design terminology

- **public methods**: methods that are part of the interface, can be called by a client (and inside the class).
- **private methods**: helper methods that can only be called inside the class.
- **private members**: variables only accessible inside the class (usually storing "state")

# Questions

Answer 2 questions.

# template class basics

# Namespaces

- Namespaces are used to put code into logical groups, so that it prevents name clashes.

- Example: I can write my own function max in my own namespace, separate from the std namespace.

- Syntax for calling/using something in a namespace

```
namespace_name::name
```

# Regular File Organization

```
// vector.h


class vector {
   void at();
}
```

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

void vector::at() {

}
```

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

int main() {
   vector vec;
}
```

Compile the .cpp and main.cpp separately.

# Template File Organization: Option 1

```
// vector.h

template <typename T>
class vector {
   void at();
}

template <typename T>
void vector<T>::at() {

}
```

Compile only main.cpp.

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

int main() {
   vector<int> vec;
}
```

# Template File Organization: Option 2

```cpp
// vector.h

template <typename T>
class vector {
  void at();
}

#include "vector.cpp"
```

Compile only main.cpp.

```cpp
// vector.cpp
template <typename T>
void vector<T>::at() {

}
```

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

int main() {
  vector<int> vec;
}
```

# Questions

Answer 2 questions.

# A closer look at the syntax

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
  if (index < 0 || index > _size)
     throw std::out_of_range("Out of bounds!");
     return _elems[index];
  }
}
```

# This is a template function

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
  if (index < 0 || index > _size)
     throw std::out_of_range("Out of bounds!");
     return _elems[index];
  }
}
```

# One template parameter T.

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
  if (index < 0 || index > _size)
     throw std::out_of_range("Out of bounds!");
     return _elems[index];
  }
}
```

# Will explain in the next section

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
    if (index < 0 || index > _size)
        throw std::out_of_range("Out of bounds!");
        return _elems[index];
    }
}
```

# This function is part of the vector<T> class, with name at.

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
   if (index < 0 || index > _size)
     throw std::out_of_range("Out of bounds!");
     return _elems[index];
   }
}
```

# Since function is a member, you can access any private members.

```
// vector.cpp

template <typename T>
value_type& vector<T>::at(size_t index) {
   if (index < 0 || index > _size)
     throw std::out_of_range("Out of bounds!");
     return _elems[index];
   }
}
```

# Summary of Template Classes

- Syntax-wise pretty much the same as functions
- You must either:
  - Put interface + implementation in the .h file (standard and very common)
  - Separate .h and .cpp, but include .cpp at the end of .cpp (not common, but we'll do it on A2 and in this class)

# Questions

Answer 2 questions.

2-min stretch break!

# Announcements

# Logistics

- Summary of Projects in CS 106L
  - Optional project 1 (GraphViz): already released
  - Optional project 2 (Interviews): already released
  - Required project 1 (Wiki Racer): due next week
  - Optional project 3 (Gap Buffer): to be released next week
  - Required project 2 (HashMap): released week 7, due week 10
  - Optional project 4 (K-d Trees): released week 8

- Each required project is around 120 lines of code

# Logistics

- Mid-quarter survey, click link on Piazza to access survey
- Anna's Wiki Racer announcement
- Office Hours - Avery has 5 hours of them today (after class), and Anna has some tomorrow!

# type aliases

# Type Aliases: basics

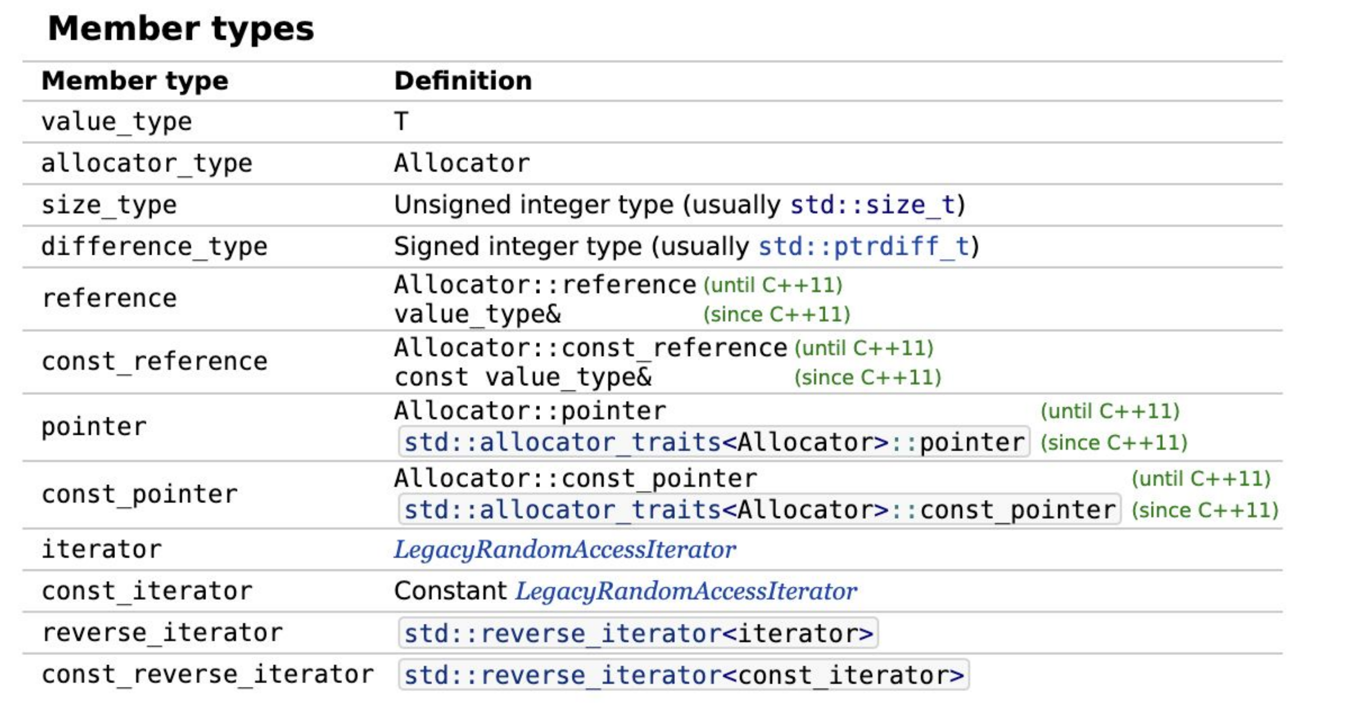
- You can give a type another name using the following syntax:

```
using another_name = existing_type;

using iterator = ????;
```

# Type Aliases: basics

- If you declare a public type alias in your class, you've created a "member type".



### Member types

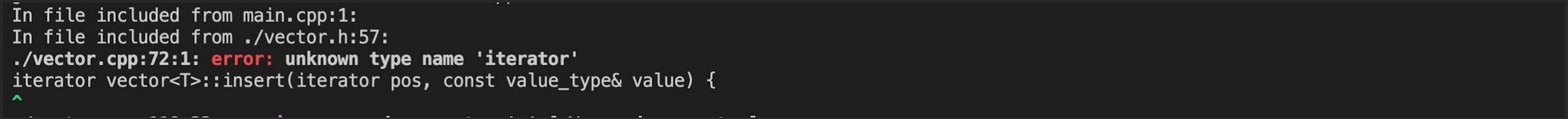
| Member type | Definition |
|---|---|
| `value_type` | `T` |
| `allocator_type` | `Allocator` |
| `size_type` | Unsigned integer type (usually `std::size_t`) |
| `difference_type` | Signed integer type (usually `std::ptrdiff_t`) |
| `reference` | `Allocator::reference` (until C++11)<br>`value_type&` (since C++11) |
| `const_reference` | `Allocator::const_reference` (until C++11)<br>`const value_type&` (since C++11) |
| `pointer` | `Allocator::pointer` (until C++11)<br>`std::allocator_traits<Allocator>::pointer` (since C++11) |
| `const_pointer` | `Allocator::const_pointer` (until C++11)<br>`std::allocator_traits<Allocator>::const_pointer` (since C++11) |
| `iterator` | *LegacyRandomAccessIterator* |
| `const_iterator` | Constant *LegacyRandomAccessIterator* |
| `reverse_iterator` | `std::reverse_iterator<iterator>` |
| `const_reverse_iterator` | `std::reverse_iterator<const_iterator>` |

# Using type alias in your .cpp file

Compiler error: can't use the name "iterator" before you've declared that you're in the vector class.

```
template <typename T>
iterator
vector<T>::insert(iterator pos, int value) {

}
```



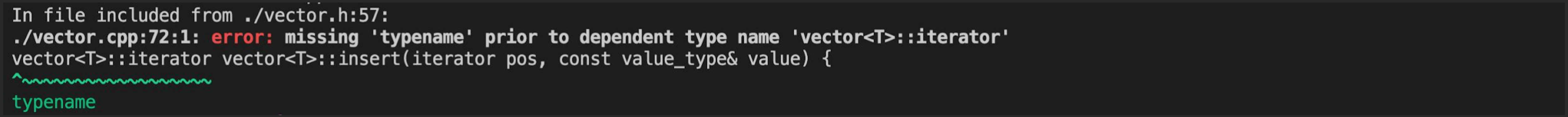
```
In file included from main.cpp:1:
In file included from ./vector.h:57:
./vector.cpp:72:1: error: unknown type name 'iterator'
iterator vector<T>::insert(iterator pos, const value_type& value) {
^
```

# Using type alias in your .cpp file

Compiler error: explanation not super important

```
template <typename T>
vector<T>::iterator
vector<T>::insert(iterator pos, int value) {

}
```



```
In file included from ./vector.h:57:
./vector.cpp:72:1: error: missing 'typename' prior to dependent type name 'vector<T>::iterator'
vector<T>::iterator vector<T>::insert(iterator pos, const value_type& value) {
^~~~~~~~~~~~~~~~~~~
typename
```

# Using type alias in your .cpp file

All good!
Literally just do what the compiler told you to do.

```
template <typename T>
typename vector<T>::iterator
vector<T>::insert(iterator pos, int value) {

}
```

# Summary of Type Alises

- Use them so your clients have a standardized way to accessing important types in your class.
- After class specifier: use alias directly.
- Before class specifier (eg. return): use

```
typename vector<T>::iterator iter = vec.begin();
```

# Questions

Answer 2 questions.

# member templates

# Member templates in classes

- Your public member functions can be templatized!
- Two problems:
  - Template declarations aren't prototypes, need to put implementation inside .h
  - Template class + template function, need to specify the template parameters for the class and for the function separately.

# Non-template classes, template member function: declare in the .h file

Declare function template inside class

```
// vector.h

class vector {
  template <typename InputIt>
  void swap_elems(InputIt first, InputIt last);
}


template <typename InputIt>
void
vector<T>::swap_elems(InputIt first, InputIt last) {

}
```

# Template classes, template member function: declare in the .h file

```
// vector.h
template <typename T>
class vector {
   template <typename InputIt>
   void swap_elems(InputIt first, InputIt last);
}

template <typename T>
template <typename InputIt>
void
vector<T>::swap_elems(InputIt first, InputIt last) {

}
```

Declare function template inside class

First class template, then function template

# Questions

Answer 2 questions.

# where we're going next

# Problems with our template class

```
void print_size(const vector<int>& vec) {
   cout << vec.size() << endl;
} // does not compile
```

const-correctness

```
void print_front(vector<int>& vec) {
   cout << vec[0] << endl;
} // does not compile
```

operator overloading

```
void make_copy(vector<int> vec) {
  // anything
} // crashes when function returns
```

copy semantics

# 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

# Next time

Const Correctness + Operator Overloading
(part 1)

there are a few small details about template classes that we don't have time to cover

A2 will walk you through them!

Supplemental Material

# Dependent Qualified Types

Dependent Name: a type that depends on template parameter

`vector<T>, T&, T::iterator`

Qualified Name: a type that appears on the right of an operator::

`T::size_type, vector<int>::iterator`

Dependent Qualified Name: a type which has a name on the right of an operator::, and the type to its left is a template type.

`vector<T>::iterator, iterator<T, IsConst>::const_reference`

# Fixing the bug

Make sure to fix the bug! Change line 58 in vector.cpp to

```
std::copy_backward(pos, end(), end()+1);
```

Reason: copy(I1, I2, I3) has undefined behavior if I3 is between I1 and I2.

Most implementations check for this, but always avoid undefined behavior!