Lecture 7: Template Functions

CS 106L, Winter ‘21
Today’s Agenda

• Recap: Iterators
• Template functions
• Announcements
• Concept Lifting
Recap: Iterators
Iterators allow iteration over any container whether ordered or unordered
STL Iterators

Generally, STL iterators support the following operations:

```cpp
std::set<T> s;
auto iter = s.begin();
++iter;               // increment
*iter;                // dereference iter to get curr value
(iter != s.end());    // equality comparison

iter = another_iter;  // copy construction
```

STL collections have the following operations:

```cpp
s.begin();            // an iterator pointing to the first element
s.end();              // one past the last element
```
Printing all elements in these collections

```cpp
std::set<int> set {3, 1, 4, 1, 5, 9};
for (initialization; termination-condition; increment) {
    const auto& elem = retrieve-element;
    cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (initialization; termination-condition; increment) {
    const auto& [key, value] = retrieve-element; // structured binding!
    cout << key << “:” << value << endl;
}
Printing all elements in these collections

```cpp
std::set<int> set {3, 1, 4, 1, 5, 9};
for (auto iter = set.begin(); iter != set.end(); ++iter) {
    const auto& elem = *iter;
    cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
    const auto& [key, value] = *iter; // structured binding!
    cout << key << "::" << value << endl;
}
```
Another option: for-each loops!

For-each loops use iterators under the hood!

```cpp
std::set<int> set {3, 1, 4, 1, 5, 9};
for (const auto& elem : set) {
    cout << elem << endl;
}

std::map<int> map {{{1, 6}, {1, 8}, {0, 3}, {3, 9}}};
for (const auto& [key, value] : map) {
    cout << key << “:” << value << endl;
}
```
Questions?
Template Functions
int my_min(int a, int b) {
    return a < b ? a : b;
}

// equivalently
int my_min(int a, int b) {
    if (a < b) return a;
    else return b;
}
Can we handle different types?

```c
int main() {
    auto min_int = my_min(1, 2);
    auto min_name = my_min("Nikhil", "Ethan");
}
```
One way: overloaded functions

```cpp
int my_min(int a, int b) {
    return a < b ? a : b;
}

std::string my_min(std::string a, std::string b) {
    return a < b ? a : b;
}
```
One way: overloaded functions

```c++
int my_min(int a, int b) {
    return a < b ? a : b;
}

std::string my_min(std::string a, std::string b) {
    return a < b ? a : b;
}
```

Bigger problem: how do we handle user-defined types? (e.g., our Student struct from a few weeks ago)
We can write a generic function!

template <typename T>
T my_min(T a, T b) {
    return a < b ? a : b;
}
Template function syntax analysis

declares the next declaration is a template

Specifies $T$ is some arbitrary type

List of template arguments

```
template <typename T>
T my_min(T a, T b) {
    return a < b ? a : b;
}
```

Note: Scope of template argument $T$ is limited to this one function!
Just in case we don’t want to copy T

template <typename T>
T my_min(T a, T b) {
    return a < b ? a : b;
}

template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}
Live Code Demo:
Templates: syntax and initialization
There are two ways to call template functions!

```cpp
template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}
```
Way 1: Explicit instantiation of templates

```cpp
template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}

my_min<std::string>("Nikhil", "Ethan");
```

Compiler replaces every T with string

Explicitly states T = string
Way 2: Implicit instantiation of templates

Compiler replaces every T with int

```cpp
template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}

my_min(3, 4);
```

Compiler deduces T = int
Be careful: type deduction can’t read your mind!

```
template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}
my_min(“Nikhil”, “Ethan”);
```

Compiler replaces every `T` with `char*`

```
Compiler deduces T = char* (C-string)
```

Comparing pointers -- not what you want!
Our function isn’t technically correct

```cpp
template <typename T>
T my_min(const T& a, const T& b) {
    return a < b ? a : b;
}

my_min(4, 3.2);
// this returns 3
```
We can specify additional template parameters!

```cpp
template <typename T, typename U>
auto my_min(const T& a, const U& b) {
    return a < b ? a : b;
}

my_min(4, 3.2);  // this returns 3.2
```

Accounting for the fact that the types could be different.
Note: Template functions are technically not functions

They’re a recipe for generating functions via instantiation.
Template Instantiation: creating an “instance” of your template

When you call a template function, either:

● for explicit instantiation, compiler creates a function in the executable that matches the initial template, with the correct template parameters

● for implicit instantiation, compiler deduces the template parameters, and creates the correct function in the same way

● After instantiation, the compiled code looks as if you had written the instantiated version of the function yourself.
Questions?
Announcements
Assignment 1 Will Be Released Tomorrow!

- Due Friday, February 19 on Paperless
- There will be a very small warm-up due next week
- We’ll send out an announcement with all logistical details
- Partners are encouraged! Check partner search capability on Piazza
Concept lifting
What assumptions are we making about the parameters?

Can we solve a more general problem by relaxing some of the constraints?
Why write generic functions?

Count the # of times 3 appears in a `std::vector<int>`.
Count the # of times “X” appears in a `std::istream`.
Count the # of times a vowel appears in the second half of a `std::string`.

By writing generic functions, we can solve all of these problems with a single function!
Remove as many assumptions as you can
int count_occurrences(const vector<int>& vec, int val) {
    int count = 0;
    for (size_t i = 0; i < vec.size(); i++) {
        if (vec[i] == val) count++;
    }
    return count;
}

vector<int> v; count_occurrences(v, 5);

🤔 What is an assumption we’re making here? (Type in the chat.)
int count_occurrences(const vector<int>& vec, int val) {
    int count = 0;
    for (size_t i = 0; i < vec.size(); i++) {
        if (vec[i] == val) count++;
    }
    return count;
}

vector<int> v; count_occurrences(v, 5);

🤔 What if we want to generalize this beyond ints?
How many times does a `<T>` appear in a vector<T>?

```cpp
template <typename DataType>
int count_occurrences(const vector<DataType>& vec, DataType val) {
    int count = 0;
    for (size_t i = 0; i < vec.size(); i++) {
        if (vec[i] == val) count++;
    }
    return count;
}

vector<string> v; count_occurrences(v, "test");
```

🤔 Perfect! But what if we want to generalize this beyond a vector?
template<typename Collection, typename DataType>
int count_occurrences(const Collection& arr, DataType val) {
    int count = 0;
    for (size_t i = 0; i < arr.size(); i++) {
        if (arr[i] == val) count++;
    }
    return count;
}

vector<string> v; count_occurrences(v, "test");

🤔 What is wrong with this? (Type in the chat.)
🚫 The collection may not be indexable. How can we solve this?
How many times does a `<T>` appear in an iterator `<T>`?

```cpp
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (initialization; end-condition; increment) {
        if (retrieval == val) count++;
    }
    return count;
}

vector<string> v; count_occurrences(arg1, arg2, "test");
```

🤔 Practice by filling in the blanks in the chat!
How many times does a `<T>` appear in an iterator `<T>`?

template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}

vector<string> v; count_occurrences(v.begin(), v.end(), “test”);

🤔 Great!
📝 We manually pass in `begin` and `end` so that we can customize our search bounds.
Live Code Demo:
Count Occurrences
We can now solve these questions...

Count the number of times 3 appears in a \texttt{list<int>}.  
Count the number of times ‘X’ appears in a \texttt{std::deque<char>}.  
Count the number of times ‘Y’ appears in a \texttt{string}.  
Count the number of times 5 appears in the second half of a \texttt{vector<int>}.  

But how about this?  

Count the number of times an odd number appears in a \texttt{vector<int>}.  
Count the number of times a vowel appears in a \texttt{string}.  
Questions?
Recap

- **Template functions**
  - lets you declare functions that can accept different types as parameters!

- **Concept lifting**
  - technique that we use to see how to generalize our code!
Next time:

lambda functions and algorithms