601.220 Intermediate Programming

Spring 2023, Day 24 (March 17th)

Today's agenda

- Exercise 23 review
- Day 24 recap questions
- Exercise 23

Announcements/reminders

- Midterm project **this evening** at 11 pm
    - No late submissions

Exercise 23 review

Read an input value into the variable count:

```
size_t count;

std::cin >> count;
```

Exercise 23 review

Make vec store count (pseudo-)random values:

```
std::vector< int > vec;

// ...

for (size_t i = 0; i < count; i++) {
  vec.push_back(::rand());
}
```

Note the #include <cstdlib> at the top of the source file. This shows how to include a C library header file in a C++ program.

Exercise 23 review

```cpp
void sort( std::vector< int > *values, int start, int end ) {
  int n = end - start;      // how many elements to sort?
  if (n < 2) { return; }    // base case
  int mid = start + n/2;
  sort(values, start, mid);
  sort(values, mid, end);
  merge(values, start, mid, end);
}

void sort( std::vector< int > *values ) {
  sort(values, 0, int(values->size()));
}
```

Exercise 23 review

Implementing `merge`: idea is to repeatly compare the smallest remaining elements from the left and right halves, and add the smaller element to the sorted result.

You'll need to use a temporary vector to hold the sorted result, and then copy it to overwrite the region being sorted.

Day 24 recap questions

1. What is a `map` in C++ STL? What is the difference between `pair` and `tuple`?
2. How do you return multiple values in C++?
3. Name some useful templated data containers provided by STL.
4. Name some useful algorithms provided by `<algorithm>`.
5. What's the difference between an `iterator` and a `const_iterator`?

1. What is a `map` in C++ STL?

`std::map` is a “dictionary” data type.

A map has two type parameters, the *key* type and the *value* type.

A map instance is a collection of pairs (k, v) where k is a value belonging to the key type, and v is a value belonging to the value type.

Duplicate keys are not allowed, so if a pair (k, v) exists in the map, no other pair in the map can have k as its key value.

Maps are very useful!

Maps have tons of uses. For example, let's say you're implementing a phone contact database, and you have the data types Name and PhoneNumberCollection.

```
struct Name {
  std::string first_name;
  std::string last_name;
};

// Name must be comparable using <
bool operator<(const Name &left, const Name &right) {
  // return true if left < right, false otherwise
}

// PhoneNumberCollection: assume this is either a struct type,
// or a typedef for some kind of collection
```

A phone database is a map of Name to PhoneNumberCollection:

```
std::map<Name, PhoneNumberCollection> phone_db;
```

Using the phone database

```
std::map<Name, PhoneNumberCollection> phone_db;
// assume that data has been added

Name n = { "Ada", "Lovelace" };

std::map<Name, PhoneNumberCollection>::iterator i =
  phone_db.find(n);

if (i != phone_db.end()) {
  // an entry for this Name exists in the map
  PhoneNumberCollection &ph_nums = i->second;

  // ...access ph_nums to get the phone numbers...
}
```

Adding an entry to a map

```cpp
std::map<Name, PhoneNumberCollection> phone_db;

Name n = { "Margaret", "Hamilton" };

// assume Name n doesn't exist in the map yet;
// using the subscript operator will add a new pair
// with n as the key and a newly-initialized
// PhoneNumberCollection
PhoneNumberCollection &ph_nums = phone_db[n];

// ...access ph_nums to add phone numbers...
```

Maps are fast!

Finding, adding, or removing a map entry requrires $O(\log N)$ time, where N is the number of elements in the map.

Log functions grow very slowly, so map lookups are efficient even when the map has a very large number of key/value pairs.

Map keys are sorted

When you traverse the pairs in a map using an iterator, you will access the keys in sorted order from least to greatest. This is a consequence of the underlying data structure, which is a balanced binary search tree.

1. What is the difference between `pair` and `tuple`? 2. How do you return multiple values in C++?

The `std::pair` and `std::tuple` types can be used to allow a function to return multiple values. (Although this is not their only use.)

An instance of `std::pair` can hold exactly two values (`first` and `second`). An instance of `std::tuple` can hold multiple values.

Note that the `std::get` function must be used to access the values in a tuple, parametized with the index indicating which value to access (0 for first value, 1 for second value, etc.)

Pair and tuple examples

```
// fruit.cpp:
#include <iostream>
#include <utility>    // for std::pair
#include <tuple>

std::pair<std::string, int> get_fruit() {
  return std::pair<std::string, int>("oranges", 8);
}

std::tuple<std::string, int> get_fruit2() {
  return std::tuple<std::string, int>("lemons", 5);
}

int main() {
  std::pair<std::string, int> fruit1 = get_fruit();
  std::tuple<std::string, int> fruit2 = get_fruit2();
  std::cout << fruit1.first << "," << fruit1.second << "\n";
  std::cout << std::get<0>(fruit2) << "," << std::get<1>(fruit2) << "\n";
}
```

```
$ g++ -g -std=c++14 -Wall -Wextra -pedantic fruit.cpp
$ ./a.out
oranges,8
lemons,5
```

3. Name some useful templated data containers provided by STL.

`std::vector`: random access sequence (like an array, but can grow)

`std::list`: sequence with sequential access (like a linked list), but $O(1)$ insertions and removals using an iterator

`std::map`: dictionary collection, maps a set of keys to corresponding values

`std::set`: sorted set of values (no duplicates allowed)

`std::deque`: first-in first-out sequence (a “queue”)

4. Name some useful algorithms provided by `<algorithm>`.

`std::sort`: sort values in any random-access sequence (array or vector)

`std::find`: sequential search of a collection

5. What's the difference between an `iterator` and a `const_iterator`?

An `iterator` allows the values in the underlying collection to be modified.

A `const_iterator` only allows the values in the underlying collection to be accessed, not modified.

iterator vs. const_iterator

Example:

```
// iter_vs_const_iter.cpp:
#include <vector>

int main() {
  std::vector<int> v = {1, 2, 3};
  std::vector<int>::iterator i = v.begin();
  *i = 42; // this is fine
  std::vector<int>::const_iterator j = v.cbegin();
  *j = 42; // compiler error
}
```

```
$ g++ -g -std=c++14 -Wall -Wextra -pedantic iter_vs_const_iter.cpp
iter_vs_const_iter.cpp: In function ‘int main()’:
iter_vs_const_iter.cpp:8:6: error: assignment of read-only location ‘j.__gnu_cx
    8 |   *j = 42; // compiler error
      |   ~~~^~~~
```

When to use const_iterator

It's always a good idea to use const_iterator in any code that is not intended to modify values in the collection being traversed.

You *must* use const_iterator when iterating over the elements of a collection accessed using a const reference. E.g.:

```
int compute_sum(const std::vector<int> &v) {
  int sum = 0;
  for (std::vector<int>::const_iterator i = v.cbegin();
       i != v.cend();
       ++i) {
    sum += *i;
  }
  return sum;
}
```

Exercise 24

- Working with strings and maps
- Talk to us if you have questions!

Hint for frequency count:

```
std::map<std::string, int> counters;
std::string word;

word = "hello";

// this works regardless of whether or not "hello" previously was
// present as a key
counters[word]++;
```

When a new key is added to a map by the subscript operator, the `second` value in the new `pair` will get the *default value* for its type, which is 0 for numeric types (including `int`).