601.220 Intermediate Programming

Summer 2022, Meeting 13 (July 8th)

Today's agenda

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

Reminders/Announcements

- HW5 is due **Thursday, July 14th**
  - We will have covered nearly everything you will need to know by the end of class today
  - File I/O will be covered on Monday

Day 23 recap questions

1. What is a template in C++?
2. What is the standard template library (STL)?
3. How do you iterate over a `std::vector` and print out its elements?
4. What is an iterator in C++?
5. How do you add an element to an existing vector?
6. (Bonus) What is the output of the program below?

1. What is a template in C++?

A template allows a `struct` type, a `class` type, or a function to be instantiated with a variety of data types or combinations of data types.

In C, a linked list node type must hard-code the payload data type, e.g.:

```
// this node type is only useful for linked lists of
// char values
struct Node {
  char data;
  struct Node *next;
};
```

C++ template linked list node type

In C++, the payload data type can be specified with a "type parameter":

```
template<typename E>
struct Node {
  E data;
  struct Node<E> *next;
};
```

Now we can have `Node<char>`, `Node<int>`, `Node<std::string>`, etc.

2. What is the standard template library (STL)?

The STL is a collection of useful template functions and classes provided by the standard C++ library.

Examples: `std::vector`, `std::list`, `std::map`, `std::sort`, many others.

Observation: to a large degree, effective programming means finding efficient and elegant ways to store, access, and do computations on data.

It is challenging to do these things in C because the only "built in" feature for aggregating data is the array, and the "built in" support for doing computations is very limited (e.g., `qsort`.)

In C++, the STL provides

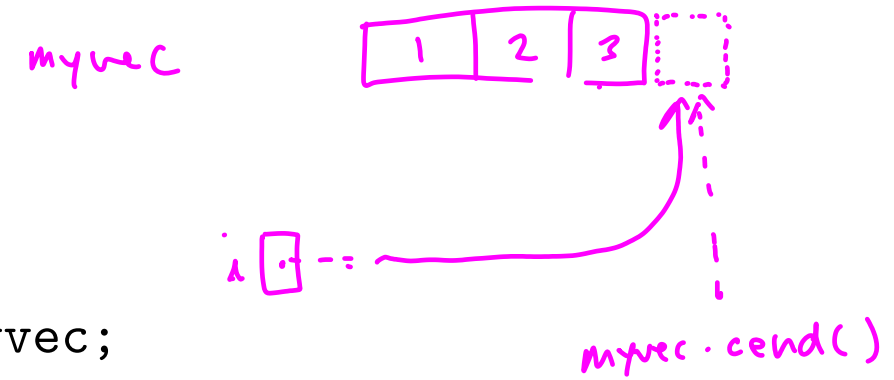
1. very powerful ways to organize and access data, and
2. powerful tools for doing computations on data

3. How do you iterate over a `std::vector` and print out its elements?, 4. What is an iterator in C++?

Traversing a collection of values in an STL container (such as a `vector`) is done using an *iterator*. An iterator is a generalization of a pointer, and is used in a way that is very similar to the way pointers are used.

In fact, a pointer to an array element *is* an iterator, because it supports all of the operations required of iterators.

Iterator example




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

// assume some values are added to myvec

for (std::vector<int>::const_iterator i = myvec.cbegin();
     i != myvec.cend();
     ++i) {
  int value = *i;
  std::cout << value << " ";
}
```

iterators

(vector) random-access i+n

(list) sequential-access i++ ++i } only

5. How do you add an element to an existing vector?

Use the `push_back` member function.

```
std::vector<int> myvec;
assert(myvec.size() == 0); // myvec is initially empty

myvec.push_back(1);
myvec.push_back(2);
myvec.push_back(3);

assert(myvec.size() == 3); // 3 elements were added
assert(myvec[0] == 1);
assert(myvec[1] == 2);
assert(myvec[2] == 3);
```

6. (Bonus) What is the output of the program below?

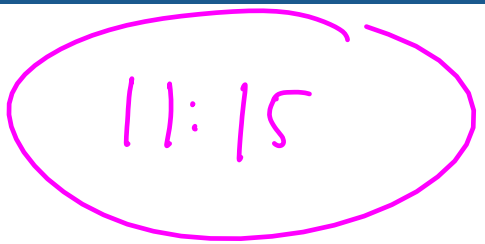
```
#include <iostream>
#include <vector>

using std::cin; using std::cout; using std::endl;
using std::vector;

int main() {
  vector<double> numbers;
  for (int i = 1; i <= 10; i++) {
    if (i % 2 == 1)
      numbers.insert(numbers.begin(), i / 2.0);
    else
      numbers.push_back(i * 2.0);
  }

  vector<double>::iterator it = numbers.begin();
  cout << "first == " << *it << endl;
  cout << "middle1 == " << *(it + 4) << endl;
  cout << "middle2 == " << *(it + 5) << endl;
  cout << "last == " << *(it + 9) << endl;
}
```

4.5 3.5 2.5 1.5 .5 4.0 8.0 12.0 16.0 20.0

— not efficient O(N)

4.5
0.5
4.0
20.0

Exercise 23



- Practice basic input and output using iostreams
- Practice using `std::vector`
- Recursion (merge sort)
- Breakout rooms 1–10 are "social"
- Use Slack to let us know if you have a question!

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?

similar to: Python dictionary, Java TreeMap

instances of std::pair<K,V> where K is key type and V is value type

`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.

Requirement: key type must be fully ordered. Default ordering is < (less than.) However, the programmer can choose an arbitrary ordering by specifying an explicit comparison functor.

Maps are very useful!

typedef some STL collection type PhoneNumberCollection;

Maps have tons of uses. For example, let's say in HW5 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
```

const reference

ordering

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 = { "Neville", "Longbottom" };

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...
}
```

reference

Adding an entry to a map

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

Name n = { "Hermione", "Granger" };

// 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

```cpp
// 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")

like vector but push-front, pop-front } O(1)

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

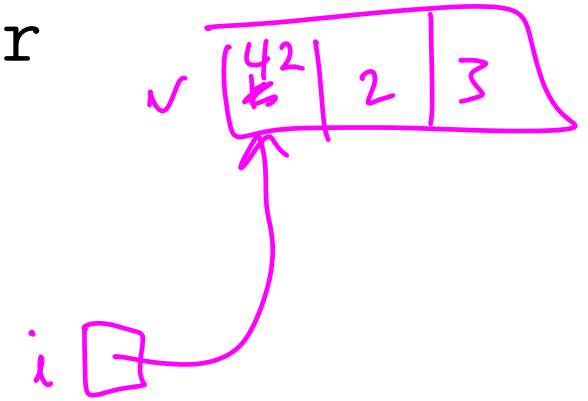
These use iterators

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 via 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
- Breakout rooms 1–10 are "social"
- Use Slack to let us know if you have a question!

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`).

Notes

Notes

Notes

Notes

Notes