CSE 333: Systems Programming

Section 7

unique_ptr

Smart pointers

- Smart pointers are an awesome feature of C++ (at least in my opinion). What benefits do they provide?
- Types of smart pointers (also shown in class)
  - unique_ptr: General-purpose container for values and arrays of values
  - shared_ptr: Reference-counted pointer. No clear ownership—prefer unique_ptr if possible
  - weak_ptr: Non-owning pointer. Can be converted temporarily to a reference-counted shared_ptr

Additional smart pointer uses

- File pointer management
  - Example: a FileCloser class that invokes fclose on the owned file pointer upon destruction
- Network socket management
  - Example: a SocketCloser class that sends an exit message and closes the socket upon destruction
- Mutex acquisition and release
  - For example: Boost’s scoped_lock

unique_ptr functionality

- Constructor takes ownership of given pointer
  - `unique_ptr<int> value_ptr(new int);`
- operator* dereferences stored value
  - `*value_ptr = 5;`
- operator= supports assignment using std::move
  - `unique_ptr<int> other_ptr = std::move(value_ptr);`
- operator-> permits access to stored value’s member variables and functions
  - `unique_ptr<string> str_ptr(new string("hello"));`
  - `size_t len = str_ptr->size();`

unique_ptr with functions

- Use std::move to transfer ownership to and from functions

```
unique_ptr<int> Multiply(unique_ptr<int> value,
                         int multiple) {
  unique_ptr<int> result(new int);
  *result = *value * multiple;
  return std::move(result);
}
...
unique_ptr<int> value_ptr(new int);
*value_ptr = 10;
value_ptr = std::move(Multiply(std::move(value_ptr), 3));
```

unique_ptr with functions

- What is stored in value_ptr before, during, and after the call to Multiply?

```
unique_ptr<int> Multiply(unique_ptr<int> value,
                         int multiple) {
  unique_ptr<int> result(new int);
  *result = *value * multiple;
  return std::move(result);
}
...
unique_ptr<int> value_ptr(new int);
*value_ptr = 10;
value_ptr = std::move(Multiply(std::move(value_ptr), 3));
```

unique_ptr with classes

```
class Example {
 public:
  inline explicit Example(unique_ptr<int> value)
    : value_(std::move(value)) {}
  inline int value() const { return *value_; }
 private:
  const unique_ptr<int> value_;
  Example(const Example&) = delete;
};
...
unique_ptr<int> value_ptr(new int);
*value_ptr = 10;
unique_ptr<Example> example(
    new Example(std::move(value_ptr)));
```

unique_ptr with STL

- Use (you guessed it) std::move when inserting or removing

```
unique_ptr<Example> example(...);
vector<Example> example_vector;
// Store the value in the vector.
example_vector.push_back(std::move(example));
...
// Retrieve the value from the vector.
example = std::move(example_vector.back());
// Remove the value from the vector
example_vector.pop_back();
```

unique_ptr with STL

- What is stored in example_vector.back() immediately prior to calling pop_back()?

```
unique_ptr<Example> example(...);
vector<Example> example_vector;
// Store the value in the vector.
example_vector.push_back(std::move(example));
...
// Retrieve the value from the vector.
example = std::move(example_vector.back());
// Remove the value from the vector
example_vector.pop_back();
```

unique_ptr and iterators

- When iterating through a container that stores unique_ptrs, use const references to the values

```
vector<unique_ptr<Example> > example_vector;
... (insert some values)
for (const unique_ptr<Example>& example :
    example_vector) {
  cout << "Value is " << example->value() << endl;
}
```

Section exercise

- Flesh out a request router (see request_router.cc)
- The request router is responsible for queuing requests as it receives them under a handler ID
- At some point, a client instructs the router to process its queued requests for a particular ID
  - Requests are removed from the queue, and resulting responses are added to the response queue for that ID
- The client can consume the list of responses for a handler ID at any point during execution
- Submit request_router.cc to the Dropbox when done