ECE 3574: C++ Idioms and Design Patterns

Changwoo Min

Updates on milestones

- **Milestone 1 grading was released.** Email me if you
  - have a question on grading
  - didn't apply patches on test files
  - got nearly 0 score due to a simple compile error
- **Milestone 2 spec is updated**
  - `memref` BNF rule and `.gitattributes`
- **Nightly build will start this Saturday**
- **Instructor's office hours**
  - Mondays 10:30AM - Noon, Fridays 2-4PM

C++ Idioms and Design Patterns

- Today we will discuss the use of design patterns and common idioms used to write canonical C++ code.
- **Common C++ idioms**
- Example: RAII, Copy/Swap, COW
- **Design Patterns**
- Example: Iterator, Factory Pattern, State Pattern, Model-View (and Model-View-Controller) Pattern

Common C++ Idioms

- All programming languages are equivalent in the sense that they are Turing complete.
- However, programming languages (or more properly the community of programmers) develop *idioms*, common ways of expressing ideas that leverages the semantics of that language.

Common C++ Idioms

- Simple example in C++: removing excess storage from a container, (e.g. a std::vector)
    - std::vector::shrink_to_fit

```
// Prior to C++11
std::vector<int>(c).swap(c);

//With C++11 (technically it is still a “non-binding request”)
c.shrink_to_fit();
```

Another simple C++ idiom: erase-remove

- What does the following print?

```
std::list<int> mylist;
mylist.push_back(0);
mylist.push_back(12);
mylist.push_back(31);
std::cout << mylist.size() << std::endl;

std::remove(mylist.begin(), mylist.end(), 12);
std::cout << mylist.size() << std::endl;
```

- Remove actually does not actually remove!

Another simple C++ idiom: erase-remove

- To really remove you use the “erase-remove” idiom.

```
mylist.erase(std::remove(mylist.begin(), mylist.end(), 12), mylist.end());
```

- See example code: `shrink.cpp`

Another simple C++ idiom: erase-remove

- Erase–remove idiom
- std::remove
- std::vector::erase

Example: RAII

- RAII stands for Resource Acquisition Is Initialization.
    - Resource acquisition is initialization
    - RAII
- See example code: `raii.cpp`

Example: Copy/Swap

- We can remove the code duplication and the self assignment test in the copy-assignment operator using the copy-swap idiom.
    - Copy-and-Swap Idiom in C++
    - Copy-and-swap
- See example code: `copyswap.cpp`

Example: Move semantics in C++11

- C++11 defines *move semantics* that add to RAII and the copy-swap idiom
    - Move semantics
    - std::move
- See example code: `copyswap11.cpp`

Example: Copy-on-Write (COW)

- A big difference between most `std::string` implementations and `QString` is the latter uses COW.
- COW is an optimization that lets objects share the same data as long as neither tries to change it, at which time a copy is made.
- Note: Matlab uses this for Matrices.
- COW has problems with concurrency, as we will see in a couple of weeks.
- See example code: `cow.cpp`

Design Patterns

- Design patterns are similar to Idioms but are less language specific.
- They are patterns in the sense of higher-order abstractions of code design.
- See the book Design Patterns: Elements of Reusable Object-Oriented Software
- There are many online compendium of patterns.

Example Design Pattern: PIMPL: Pointer-to-Implementation

- Pimpl decouples the definition and implementation of a class stronger than via private and public.
    - Pointer To Implementation
    - PImpl
- Can be usefull for abstracting platform differences without headers full of macros.
- Qt uses the Pimpl pattern extensively.
- See example code: `pimpl/*`

Example Design Pattern: Iterators

- Iterators are used throughout the standard library for accessing and manipulating containers.
- They are an abstraction of pointers.
- See example code: `iterators/*`

Criticisms of Design Patterns

- To some extent the patterns are ways of expressing things not naturally found in the language.
- Some people consider this a limitation of the programming language in question.
- It is easy to go overboard. Some patterns are overused (in my opinion), Singleton for example.

Design Pattern: Factories

- When using dynamic polymorphism it is common to have many types that derive from a common base with the subtype specified at runtime.
- The Abstract Factory pattern is a class that builds subtypes based on a description, usually derived from user input, and returns a base pointer to the constructed object. This collects switch-based object construction code into one place.
- See example code: `shape_factory.cpp`
- Note, this works best when using the object does not require casting (as in good polymorphic design).

Remember destructors for base classes should be virtual

- The example I showed used stack allocated objects inside the derived class.
- If you have to manage resources in the derived class, make sure the base class has a virtual destructor
- See example code: `base_virt_dest.cpp`
- Reference: Factory method pattern

Design Pattern: State (as in State Machine)

- The state pattern uses a private pointer to a state object to encapsulate behavior based on the state an object is in, with the ability to transition.
- See example code: `code/state/*.cpp`
- This is useful whenever you need to cleanly code a complex state machine.
- Reference: State pattern

Model-View Pattern

- The model-view pattern separates data (the model) from the code used to present it (the view).
- Communication happens through a well-defined interface.
- Thus any object that conforms to the interface can be viewed without custom code.
- See example1 and example2 code.

Model-View-Controller Pattern

- For interactive applications (e.g. GUI) it is common to introduce a third object called the controller that mediates between user actions, the view, and the model.
- Typically in Qt the models, views, and controllers communicate using a mixture of events and the signal/slot mechanism.
- See example3 code.
- References
    - Model–view–controller
    - QFileSystemModel

Next Actions and Reminders

- Read Chapter 26 of Operating Systems: Three Easy Pieces