CS 247 Midterm Review

CS 247

University of Waterloo

cs247@uwaterloo.ca

June 18, 2018
Overview I

1. Intro
2. ADTs
   - Benefits to ADTs
   - Outside-In Design
   - ADT Design Choices
   - Entity-Based vs Value-Based
   - Special Member Functions
3. Modularization
4. Documentation
   - Interface Specification
   - Representation Invariants
   - Abstraction Function
5. Header Conventions
6. Exceptions
7. Resource Acquisition Is Initialization (RAII)
8. Iterators
Overview II

9 UML Models

10 Design Patterns

- Template Method
- Observer
I have **not** seen the midterm; this is a general review of the material we have seen so far.

**Details:** Thursday June 21st, 4:30-6:20PM, MC 1056 or MC 1085

**Coverage:** All material from before the midterm.

**Resources:**
- lectures (inc. code)
- tutorials
- assignments
- design patterns webpage
Benefits to ADTs:

- Limits what client needs to know in order to use the class
- Maintains representation invariants and encapsulation
  - Limits the need for checks
- Improves code evolvability; we can change the implementation without the client code needing to change
Outside-In Design

Be familiar with the steps of outside-in design and be able to apply them to a “problem”:

1. Use scenarios
2. Interface design
3. Implementation
4. Validation
   - return to #1
ADT Design Choices

How do we make decisions about how to design our ADT? Specifically:

- Should our functions be members? friends? neither?
- Should our parameters be a value? a reference? const?
- Should we use default arguments or function overloading?
- What declaration keywords do we need to use?
  - virtual
  - override
  - explicit
  - const
  - final
Entity based ADTs usually:

- Prohibit assignment and the copy constructor
- Use inheritance
- Avoid equality
- Are mutable

This is because entities are usually unique - they represent a single object in the real world.
Value based ADTs usually:

- Implement equality and other comparison operators
- Include a copy constructor and assignment operator
- Are immutable (instead of changing the value a new value is created)

Oftentimes, a Value-based ADT can package together multiple primitive values (e.g. Rational Numbers) and/or add some restrictions on values (e.g. denominators can’t be zero, account numbers that must be unique).
Mutable vs Immutable

Whether to make an ADT mutable or immutable is often decided by whether it is entity-based or value-based. Mutable ADTs should include functions that allow them to be mutated while immutable ones should not - to use a different value, you typically just construct a new value.
In some cases, whether an object is entity-based or value-based is not 100% clear. An example of this is the Collection ADT from Assignments 1 and 2. A Collection is mutable (because users can be added to and removed from it), which implies that it is an entity; however, if two collections contain the same users, you would consider them identical (this is not usually the case for an entity-based ADT).
C++ special member functions are functions that the compiler will generate for you under certain conditions.

The six special member functions are:
- Default Constructor
- Destructor
- Copy Constructor
- Copy Assignment Operator “=”
- Move Constructor
- Move Assignment Operator “=”
Special Member Functions

Make sure you know the behaviour of compiler generated functions; i.e. shallow vs. deep copy

Make sure you when special member functions are being generated by the compiler. See the special members table.
“Rule of 5” functions:

- Destructor
- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator

The rule of 5 states that if we define any of these 5 functions ourselves, then we should likely define all 5 of them ourselves.
Copy & Swap Idiom

For a short and clean implementation of the copy assignment operator we can use the **Copy & Swap Idiom**.

```cpp
myClass & operator= (const myClass & other) {
    myClass temp(other);
    swap(*this, temp);
    return *this;
}
```

Why does it work
Benefits of modularization:

- Allows development both independently and incrementally
- Allows for separate compilation
- One module’s implementation can change as long as it still conforms with its interface
- Higher reusability of components

Benefits of header files:

- Abstract view of module; useful for both clients and coworkers
- Ensures consistency
Thoughtlessly including lots of files can lead to cyclical dependencies. But we can break out from them by using forward declarations. But when should we use what?

Use forward declaration when no true compilation dependency exists. See the forward declarations page to test yourself (you can ask for answers on Piazza)
plmpl Idiom

Why do we use the plmpl Idiom? Used to hide the implementation from the client, which is beneficial because:

- allows us to change the implementation without need for the client to recompile
- makes the interface simpler; hides irrelevant details

How do we use it?

- myImplClass * pImpl;
- myImplClass is either a nested class or a friend class
- **remember**: it is still your implementation, it is not breaking encapsulation to allow it full access to your class
Interface Specification

The essential components of an interface spec are:

- specification fields: the client programmer’s abstract view of an object’s fields
- requires: necessary pre-condition assumptions about the program state
- modifies: list of objects and/or variables that may be changed
- throws: exceptions that are thrown, and in which conditions
- ensures: how the objects and/or variables from ‘modifies’ are changed
- returns: the return value
Some good practices that we should follow when writing our specs:

- preconditions and exception conditions should be disjoint
- specs are considered in order; ‘throws’ comes before ‘ensures’
- use Abstract Specification Data Members
  - ex. in a2q1, binary_str
The Specificand Set of a specification is the set of all conforming implementations.

We want our spec to restrict unacceptable implementations (ie. set with duplicates) and be general enough to allow several different implementations (ie. array or linked list implementation of sets).

Should we check preconditions in our code?

Review how to compare two different specs.
A Representation Invariant is a constraint that characterizes whether a concrete value of an ADTs implementation is well-formed, from a representation point of view.

For example: restricting values for certain variables, pointers that cannot be nullptr, items in a collection must be unique, etc.

Similar to specs, you can/should use Abstract Spec Data Members to make writing representation invariants easier and to hide some of the unnecessary implementation details.
Representation Invariants

We can rely on our invariants to help us write cleaner, more efficient code by limiting the number of checks we need to do (i.e. nullptr checks).

Try to think of representation invariants that:

- preserve the logic of whatever you are implementing (i.e. no duplicates in a set)
- guarantee legal values for a field (i.e. denominator_ != 0)
So what causes representation exposure?

- non-private data members
- client has access to internal representation; for example, client holds a reference to an object that is passed in to a method and that object is incorporated directly, instead of copying its values
- operation outputs a reference to internal representation

When using our checkRep() function, why is it a good idea to put it at the **start** of our mutators, as well as at the end?
An abstraction function of an object “r”, written as $AF(r)$, explains the correlation between concrete values in an ADTs implementation and all of the ADTs abstract values.

Typically this is done by collecting individual elements (in the case of a container) or with a function that maps values from one kind to another (e.g. a number between 0 and 51 to a card in a deck). These will typically use a combination of code, math, English, and examples.
Let’s say we wanted to create an ADT that represented a Set; what would be the Interface Specification for the “void insert(int)” and “void delete(int)” functions?
// spec fields: setMembers: all items in the set

void insert(int target):
// requires: target is not already in setMembers
// throws: none
// modifies: setMembers
// ensures: setMembers = setMembers@pre with target added
void delete(int target)
// requires: target is in setMembers
// throws: none
// modifies: setMembers
// ensures: setMembers = setMembers@pre with target removed

What if we wanted a more robust Set, which threw an exception if you tried to insert a duplicate element or delete a non-existing one?
void insert(int target):
// requires: none
// throws: DuplicateElemException,
// if target is already in setMembers
// modifies: setMembers
// ensures: setMembers = setMembers@pre with target added
void delete(int target)
// requires: none
// throws: NodeNotFoundException,
// if target is not in setMembers
// modifies: setMembers
// ensures: setMembers = setMembers@pre with target removed
Let’s say we were going to implement our set using a Binary Search Tree. What would be a BST-Set’s Representation Invariant and Abstraction Function?
BST-Set Representation Invariant

RI(s):
    for all nodes n in s,
    n.key > n->left.key AND
    n.key < n->right.key
BST-Set Abstraction Function

AF(s):
    if s is null
        return {}

    return \{s.key\} \cup AF(s.left) \cup AF(s.right)
Header Guards

Header guards prevent errors from the same things being defined multiple times when files are included in more than one other file in a project.

```cpp
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass {
    ...
};
#endif
```
Do not include “using” directives in header files. When you do so, any files that include the header will also have the effect of “using” which can lead to unexpected name clashes.
Exceptions allow us to separate error-handling code from normal code and prevent errors from being ignored. They propagate errors that the current function can’t handle to a place where it can be handled (ex. BinaryBum::toDecimal()).

Include libraries exception and stdexcept. All exception objects (whether from a library or self-made) should have a “const char * what()” function that returns information about the exception.
Throwing Exceptions

A exception thrown within a ‘try’ block searches for the “nearest” matching ‘catch’ statement, unwinding the call stack as it goes.

This is fine for stack allocated objects, but could cause memory leaks from heap-allocated objects (we will solve this with RAII).

We can re-throw an exception by just using ‘throw’. If we want any changes to a re-thrown exception to persist we need to originally catch the exception by reference.

Catch statements for a base type exception can catch a derived type exception (IS-A relationship). See exception.cc.
A function is **exception-safe** if it leaves the program in a “valid state” after terminating by throwing an exception:

- **basic guarantee**: invariants of all objects are maintained and no resources are leaked
- **strong guarantee**: the program state will be exactly as it was had the function not been called
- **no-throw guarantee**: the function will never throw or propagate an exception; it always completes its task

See `exceptionSafety.cc`.
RAII is the principle that the acquiring of a resource **is** the initialization of a statically allocated object whose job it is to manage that resource. So resource acquisition **is** initialization because the acquisition of a resource should only be done when initializing a statically allocated object.

```cpp
class Resource {
  resource_type *r_;
  resource_type* allocate( parms p );
  void release ( resource_type* );
public:
  Resource (parms p) : r_( allocate(p) ) {}  
  ~Resource() { release(r_); }
... 
};
```

Make sure to be familiar with smart pointers.
Iterators

To use an iterator we need:

class A (iterator):
  - prefix operator++
  - operator!=
  - unary operator *

class B (main class):
  - A begin()
  - A end()

In practice, the iterator class is often simply nested inside the main class.
Iterator Details

Make sure to be familiar with the ‘auto’ keyword and how it helps simplify the use of an iterator.

We can also use the range for loop notation:

```cpp
for (auto n : collection) { ... }
```

Remember though, we need all of the necessary functions (see last slide) for this notation to work.

Suggestion: if you didn’t originally, go back and use an iterator on a2q2 for practice
Benefits of using models:

- Increases understanding of program before modeling
- Able to clearly communicate to others how your program works
- Able to answer questions more easily

For example, the multiplicities in a UML model can help us determine how our class will be implemented (see multiplicities example)

Suggestion: create class models, object models, and/or sequence diagrams for programs we have made in lecture, tutorial, or on assignments.
Class Model Relations

**Association:** use a line or an arrow to indicate an association between two classes.

**Association Classes:** used to describe the association between a pair of classes as a class.

**Composition:** A ‘owns-a’ B

**Aggregation:** A ‘has-a’ B

**Generalization:** B ‘is-a’ A
An object model is a run-time instance (snapshot) of a class model.

Objects are instantiations of classes and links are instantiations of associations.

Suggestion: draw a class diagram from a provided object model or vice versa.
A sequence diagram is a model of communication events between objects as exhibited in one execution trace.

It not complete (exhaustive) and it is not absolute (ordering of function calls could change slightly).

See sequence diagram example.
Template Method

When to use the template method pattern?
We want subclasses to override some aspects of superclass behavior but other aspects must remain the same.

The template method pattern promotes code reuse by providing a skeleton of code for an algorithm while deferring some steps to subclass methods.
What are two essential components in the observer pattern?

One object, the subject, maintains a set of dependents, the observers. When the state of the subject changes it notifies the observers which then pull the state of the subject to get the relevant information.

See the observer class model and sequence diagram.
Q&A