Overview

1. ADTs
   - Default Arguments
   - Overloading
   - Special Member Functions
   - Entity-Based vs Value-Based

2. Documentation
   - Interface Specification
   - Representation Invariant
   - Abstraction Function

3. Exceptions

4. Header Conventions

5. Resource Acquisition Is Initialization (RAII)

6. Design Patterns
   - Template Method
   - Observer
   - MVC
A default argument is a value provided in the function declaration that is automatically assigned by the compiler if caller of the function doesn’t provide a value for the argument with default value.

Arguments with default values cannot be followed by arguments without default values.
What’s wrong if we declare the following two functions in the same program?

```c
void test (int n, string foo);
void test (int n = 0, string foo = "bus");
```
Function overloading:
Functions arguments differ in number or types.
operator+
Add two numbers: int operator+(int, int);
String concatenation: string operator+(string, string);
The seven special member functions are:

- Default Constructor
- Destructor
- Copy Constructor
- Copy Assignment Operator "="
- Move Constructor "="
- Move Assignment Operator
- Equality Operator "=="
When designing an ADT, one of the first questions that must be asked is whether the ADT is entity based or value based.
Entity-Based

Entity based ADTs usually:

- Prohibit assignment and the copy constructor
- Prohibit type conversion
- 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).
The interface of an ADT (i.e. the methods available to the client programmer) can have some documentation that briefly and succinctly tells the programmer what each function does. The essential components of these are:

- **specification fields**: the client programmer’s abstract view of an object’s fields
- **requires**: pre-conditions that must be true
- **throws**: exceptions that are thrown, and in which conditions
- **modifies**: list of members that are changed
- **ensures**: how those members are changed
- **returns**: self-explanatory
A Representation Invariant is a predicate - a condition that can be evaluated to “true” or “false”. For an ADT to be considered valid, its representation invariant must be true at all times. For example: restricting values for certain variables, pointers that cannot be nullptr, items in a collection must be unique, etc.
An abstraction function of an object “r”, written as $AF(r)$, is a way of telling the reader how to get from the concrete implementation of an ADT to the abstract, conceptual version of the ADT. 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, and English.
Let’s say we wanted to create an ADT that represented a Set, and we were going to do that using a Binary Search Tree. What would be the Interface Specification for a BST’s “void insert(int)” and “void delete(int)” functions? What would be a BST-Set’s Representation Invariant and Abstraction Function?
BST-Set Interface Specification

// 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
BST-Set Interface Specification

```c
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 BST-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
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 empty
    return Union of {s.key} AND
        AF(s->left) AND
        AF(s->right)
Exceptions allow us to separate error-handling code from normal code and prevent errors from being ignored. The best practice when creating exceptions is usually to create an exception class that inherits from std::exception and implements

\texttt{virtual const char* what()}

You can also create your own exception objects. Oftentimes, the name of the class itself is enough to tell you what the exception is, though the more information you include, the better.
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.

```c
#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.
Resource Acquisition Is Initialization (RAII) is the idea that you should encapsulate a resource (an input stream, network connection, etc.) as an object, so that the resource is allocated in the constructor, and gotten rid of in the destructor. This frees the programmer from the responsibility of releasing resources themselves.

```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_); }
  // accessors, mutators, etc.
};
```
The design patterns covered were:

- Template Method
- Facade
- Adapter
- Strategy
- Observer
- Model-View-Controller
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 is the facade pattern?
The facade pattern is used to provide a simplified interface to a complex body of code. This is accomplished through a wrapper class that contains all of the desired members.

Original system is still accessible.
What is the adapter pattern?
Adapter

The adapter pattern is a way of allowing the interface of an existing class to be used by another interface. A class is defined that translates between the two incompatible interfaces.
What is the strategy pattern?
The strategy pattern is a way of making algorithms interchangeable at run-time. This is accomplished by providing an abstract base class as an interface for the algorithm and creating concrete derived classes with different implementations of the algorithm.
What are two essential components in the observer pattern?
Observer

One object, the subject, maintains a set of dependents, the observers. When the state of the subject changes it notifies the observers usually by calling one of their methods.
What is MVC?
Model-View-Controller

MVC is used when designing user interfaces, separating the application into three parts: model, view and controller.

- The model takes care of the logic of the application, updating the view when it changes.
- The view uses the observer pattern to interact with the model, receiving a notification, and then requesting information from the model.
- The controller takes user input and sends commands to the model.
Q&A