Today ...

- Course overview
- Basic C++ Inheritance
- Homework 1
- Reading assignments
  - Appendix A
  - Sect. 1.3
  - Ch. 3
  - Handout (optional)
About this Course

Instructor
Shawn Bowers
– Email: bowers@gonzaga.edu
– Office: 309C Herak
– Office Hours: M-Th, 3:00pm–4:00pm (or by appointment)

Course webpage
– http://www.cs.gonzaga.edu/~bowers/courses/cpsc223
– Grades posted to http://blackboard.gonzaga.edu

Textbook
– Data Abstraction & Problem Solving with C++: Walls and Mirrors, 5th edition, Carrano, Addison-Wesley, 2006

About this Course

Continuation of CS II, emphasis on Data Structures, ADTs, and Algorithm Analysis

More C++ programming

Grades:
– (35%) Individual Assignments
– (10%) Group project
– (15%) Quizzes
– (20%) Two Midterm Exams
– (20%) Final Exam

Letter grade cutoffs  A: 90-100%, B: 80-89%, etc.
About this Course

Reading assignments
– Due before next class period

Homework & project assignments
– Programming assignments (done individually)
– One small class project (within a group)
– Please turn in assignments by the due date (see syllabus)

Exams and quizzes
– Open book and note
– Two Midterms and a Final exam
– Short quizzes at the beginning of classes (approx. 10)

About this Course

Attendance
– Participation is an important part of this course
– 4 absences without prior arrangement will lower your grade by one letter (as will each subsequent 2 absences)
   ※ Unless you’re sick (especially if it resembles the flu!!!), then please stay home!

Academic Honesty
– You are expected to follow the CS Department Policy
– Assignments, exams, quizzes done individually
– Group projects to be done by the group
– Please ask if you have questions or concerns
About this Course

No classes …
   – Thursday, November 26 (Thanksgiving)

Midterms and Final
   – Midterm I, Sept. 30 (in class)
   – Midterm II, Nov. 4 (in class)
   – Final, Dec. 16 (8:00am – 10:00am)

Be sure to carefully read the syllabus

Course Topics

• Abstract Data Types & Data Structures

• More C++ and Object-Oriented Programming
   – Inheritance, operator overloading, virtual functions, friends
   – Heavy emphasis on dynamic structures (i.e., pointers!)

• Data structures
   – Linked lists, binary search trees, balanced trees, hash tables

• Algorithms
   – Searching and sorting
   – Analysis using big O notation
Course Topics

• Abstract Data Types & Data Structures

• More C++ and Object-Oriented Programming
  – Inheritance, operator overloading, virtual functions, friends
  – Heavy emphasis on dynamic structures (i.e., pointers!)

• Data structures
  – Linked lists, binary search trees, balanced trees, hash tables

• Algorithms
  – Searching and sorting
  – Analysis using big O notation

This class is tough!

• Lots of programming
  • Some hard assignments

Give yourself plenty of time

First Quiz

• This is an easy one …

• Why quizzes?
  – To see the types of questions I ask on exams
  – To give feedback on understanding of material
  – To help you master the material

The main way to do well on quizzes is to take them!
Introduction to ADTs & Data Structures

Why study ADTs and Data Structures?

Supported in many “modern” programming languages

- **Java**: Collections API (List, Set, Queue, Stack, Map, …)
- **Python**: Primitive constructs (Lists, Dictionaries, …)
- **Even C++**: Standard Library (List, Set, Stack)

As a programmer, it is good to understand

- Different structures and how they are implemented
- Performance trade-offs

A major topic in computer science & programming

- Choose efficient structures to specify and implement algorithms
- Generic and reusable software components (toolbox)
- Used in almost all software applications

In this course …

- We’ll focus on
  - different ADTs and data structures
  - algorithms to solve problems (searching, sorting) based on these
  - ways to measure the efficiency of approaches used

- This course is **not** just about learning new C++ syntax

- Instead, we’ll apply C++ to implement new structures
  - as a side effect we’ll look at new C++ language features

- Create **reusable** software components (structures)
## Some Key Terminology

- Modularity
- Data Abstraction
- Abstract Data Types
- Data Structure
- Client and User

## What we mean by “Modularity”

**Modularity** is a design principle for managing the complexity of programs

- Build software out of smaller, self-contained pieces
  - Modules may already exist (reuse)
  - New modules can be specified early
  - New modules can be implemented independently (even if other modules of the program are not implemented yet)
What we mean by “Data Abstraction”

Data abstraction is a design principle that separates the allowable operations on a collection of data from the implementation of these operations.

- Provides a separation (a “wall”) between a high-level definition and the concrete implementation
  - program against the high-level definition
  - focus on “what” can be done instead of “how” it is done
- Allows changing implementation without changing dependencies

What we mean by “Abstract Data Type” (ADT)

An Abstract Data Type (ADT) is a mechanism for realizing data abstraction.

- An ADT consists of a collection of (allowable) data and a set of operations on that data
  - Together they specify how a “data type” behaves
  - Ideally, all interaction with the data is only through ADT operations
  - The ADT specifies a “contract” with the client
  - “Data type” is broad … e.g., primitives like ints, doubles, strings, or structures like arrays, lists, trees
An Aside: ADTs as Algebras

An algebra is typically specified by giving:

- The “sorts” (a set of types, e.g., \{Integer, Boolean\})
  - Sometimes called the “carrier sets” (for the sets of the types)
- Function signatures (syntax)
- Axioms characterizing the functions (semantics)

<table>
<thead>
<tr>
<th>Carriers:</th>
<th>A, Boolean, lists(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations:</td>
<td></td>
</tr>
<tr>
<td>isEmpty: lists(A) \rightarrow Boolean</td>
<td></td>
</tr>
<tr>
<td>cons : A \times lists(A) \rightarrow lists(A)</td>
<td></td>
</tr>
<tr>
<td>head : lists(A) \rightarrow A</td>
<td></td>
</tr>
<tr>
<td>tail : lists(A) \rightarrow lists(A)</td>
<td></td>
</tr>
<tr>
<td>Axioms:</td>
<td></td>
</tr>
<tr>
<td>isEmpty (&lt;&gt;) = true</td>
<td></td>
</tr>
<tr>
<td>isEmpty (cons(x, L)) = false</td>
<td></td>
</tr>
<tr>
<td>head(cons(x, L)) = x</td>
<td></td>
</tr>
<tr>
<td>tail(cons(x, L)) = L</td>
<td></td>
</tr>
</tbody>
</table>

f: X \rightarrow Y a function from the set X to the set Y

A set of values (type variable)

From these operations (with +, =) various others can be defined (e.g., length, member, index, last, first, ...)

What we mean by a “Data Structure”

A Data Structure is a specification of how data collections are stored

- For example, an array or a linked list
- The data of an ADT is stored in a data structure

- ADT ! = Data Structure
  - List is an ADT (the what)
  - Linked list is a data structure (the how)

- In C++, we can
  - Use built in data structures, e.g., arrays and structs
  - Create our own from these (e.g., linked list of structs)
What we mean by “Client” and “User”

A Client is the program or module that uses a module

- Sometimes referred to as a “dependency”
- A program is a client of the modules it uses
- A Queue implementation may be a client of a linked-list data structure

A User is (typically) a person that executes and interacts with a program

- We often call the program the “user application”
- “User app” and “Client program” used interchangeably

In this course, we mainly focus on

Modular design through ADTs and Data Structures …

- Use data abstraction
- Learn (more of) the “standard” ADTs
  - lists, trees, tables
- Learn (more of) the “standard” data structures
  - dynamic memory, pointer-based implementations
- Analyze these approaches
  - algorithm efficiency (theoretical)

Generic C++ components (re)used in many apps
Aside: Theoretical vs. Empirical Evaluation

We mainly discuss theoretical efficiency of algorithms

– For example, upper bounds on algorithm complexity
– Independent of data, machine, or implementation used

The group project will focus on empirical (experimental) efficiency of implementations

– How do approaches perform on actual data?
– Analyzing algorithm complexity is often much harder
– Experimental “benchmarking” is very common in CS

We’ll talk more about the project later in the course

Object-Oriented Programming

In this course, what is “good” C++ software design?

1. Use C++ classes to define ADTs
2. All data members private
3. All ADT operations public
4. Client should not be aware of data structure used
   – For example, that there is a node or next pointer in a linked list, or an index to an array
   – Allow an ADT to “plug and play” different data structures to improve efficiency without disrupting client program
Object-Oriented Programming

In this course, what is “good” C++ software design?

5. For each data member, determine if it could be a local variable to a member function instead
   – If the value of the variable does not need to persist from operation to operation, it should not be a data member
   – Restrict data members to what is necessary

6. Classes should not prompt for input
   – The ADT should be as general as possible …
   – … should not tie member functions to reading information from the keyboard

Object-Oriented Programming

So in this course, what is “good” C++ software design?

7. Statically allocated arrays should not be used to implement ADTs
   – Instead, use dynamically allocated arrays … why?
   – Allows the ADT to be used by many applications
   – The main program is the only place you should use statically allocated arrays
Object-Oriented Programming

In this course, what is “good” C++ software design?

8. Our client programs represent
   – a “test bed”
   – main function should handle I/O, create objects, call and test methods

9. Well-designed tests to exercise ADT operations
   – tests should help to ensure reusability
   – test range of data, including end points
   – test invalid data
   – as reasonable, use files for input

10. No global variables

11. Focus on efficient implementation
   – E.g., only traverse structures when absolutely necessary
   – Use pass-by-reference to reduce instances of structs and classes from being copied
Programming Assignments

• Managing a simple *Electronic Dictionary*
  – Consisting of words and definitions
  – We will develop different ADTs and algorithms to *store*, *search*, and *sort* dictionaries

• The basic types (modules) of our dictionaries are
  
  **Key**: a class to represent dictionary words

  **Entry**: a class to represent definitions of words

  **Dictionary**: a class to represent a collection entries

Assignment 1

• Implement a Key and Entry class
  – Entry is a subclass of Key
  – Do basic operator overloading (==, <, ==)
  – Hint: use the C++ string class
Assignment 1

Some advice for streamlining your coding

– Make sure submit works for you (and submit often!)
– Try developing your code on ada (e.g., using pico, vi, or emacs)
– Write your code incrementally!!!

```c
while(!doneWithProgram) {
    writeSmallChunkOfCode(); // e.g., one function
    compileErrors = compileProgram();
    while(compileErrors != null) {
        fixCompileError(compileErrors[0]); // first error first!
        compileErrors = compileProgram(); // recompile
    }
}
```

For Assignment 1, we’ll cover

• Basic inheritance (... today)
• The various uses of the const keyword
• Basic operator overloading
  – operator==
  – operator<
  – etc.
• The “this” keyword
• Assignment operator and copy constructor

We’ll go over more details later