Announcements

- Program #3
  - Due Tuesday at 8pm
- Reading
  - Chapter 17 (Today)
    - skip 17.5.4
  - Chapter 13 (Thursday)

Implementation Hiding

- Goals
  - Export a public API
  - Hide details of implementation
    - allows implementation to evolve
    - better able to reason about a large system
      - don't need to know how it works, just that it works
    - protects IP in implementation
  - Get the right abstractions
    - This is the hardest part!
- Techniques
  - No implementation code in header files
  - Careful use of pointer definitions
    - use opaque pointers

Opaque Pointer Types

- Generic Pointers
  - make API return **`void*`** pointers
  - lacks type checking
  - useful for building containers
    - such as maps, sets, etc.
  - Example:
    - void *createTable();
    - addItem(void *table, void *data);
- Pointers to structs (struct defintions are hidden)
  - Public header files define API and pointer to abstract types
    - API users only see public header files
  - Private header files define actual structures
    - Include public header files
    - Visible to modules implementing the API

Information Hiding Example

- Project #3 revisited
  - (but don't change project 3 – this is just another model)
- Change ast.h (to be the public header)

```
struct   _node;
typedef    struct   _node    ASTnode;
```

- Create astP.h (private header)

```
struct    _node {
    nodeType type;
    struct   _node *left;
    struct   _node *right;
    int value;
    operatorType operator;
    char *name;
};
```

Containers vs. APIs

- Containers
  - designed to hold a collection of the same or similar objects
  - provide common access to stored items
- API
  - encapsulates a useful bit of functionality
  - Example: AST's in project #3
  - May use containers to store information

Common Container Abstractions

- Stack
  - Last-in/first-out semantics (LIFO)
  - Push and pop are operations
- Queue
  - First-in/first-out semantics (FIFO)
  - Enqueue and dequeue are only operations normally
- Table (really a variation on a map)
  - Storage of keyword,values
  - Supports insert, lookup, delete

Table Container

- Can have many underlying implementations
  - Arrays
  - Linked list
  - Hash table (open or chained)
  - Trees (many different trees possible)
- Each implementation has performance attributes
  - How much space is required ?
  - How efficient is insert vs. lookup vs. delete ?
  - Implementation time vs. amount of time/space saved ?
- No single right implementation for all purposes
  - Project #4 explores two alternative implementations

Designing a Good API

- Why it matters:
  - Can always change a bad implementation
  - Changing an API impacts all users
- Elements of a good design
  - Can be easily used for many different purposes
  - Focus on key ideas, not how they work
    - Lookup/Store data **NOT** insert into tree
  - Logical units of work with each function call
  - Simple (and common) things should be easy
    - add convenience functions if needed
      - functions that can be expressed using others in API
- Elements of a bad design
  - Exposes aspects of the implementation
  - Abstractions don't match the common usage pattern

A Bad API Example

```
typedef struct {
    float real;
    float imaginary;
} complexNumber;

complexNumber *createComplex();
float getRealPart(complexNumber *num);
float getImaginaryPart(complexNumber *num);
void setNumber(complexNumber *num, float real, float imaginary);
```

- Hides nothing
- Provides little (to no) value added beyond struct

More Elements of good API Design

- Consistent Return Values across functions
  - 0 is always success
  - negative is always failure
- Consistent parameters
  - use the same names for the same things in different functions
  - how copy semantics are handled
    - are strings copied when passed to API