A Fable

(by John C. Reynolds, 1983)

Once upon a time, there was a university with a complex number policy. All faculty were required to maintain a library catalog system. In one section, Professor DesCartes announced that a complex number was an ordered pair of reals, the real section was nonnegative, and that two complex numbers were equal if and only if both their components were equal. Despite this, neither DesCartes nor Bessel ever committed moral turpitude.

In the other section, Professor Brian announced that a complex number was an unordered pair of real numbers, one nonnegative, and that two complex numbers were equal if and only if both their components were equal. DesCartes or Bessel's definitions were equal (other than that) not a single line of code had to be changed.

In the first section, the team that was using the linked symbol table implementation, realized that their performance was slow on large datasets: \(O(N)\) time. They announced that a symbol table was a linked list of operations upon them, thereafter they spoke at a conference.

In the other team, Dr. Dondero announced that a symbol table was an array of linked lists, indexed by a "hash" value. He then went on to define "put" and "get" operations on symbol tables.

In the first section, the team leader Dr. DesCartes went on to define "put" and "get" operations on symbol tables. He then told an entirely different story about converting reals, "C" addition, multiplication, conjugation and magnitude.

Retelling the Fable

Finally, the team that was using the linked symbol table implementation, realized that their performance was slow on large datasets: \(O(N)\) time. They simply calculated the hash code implementation, and, other than that, they saw a single line of code had to be changed.

"Programming in the Large" Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms (done)
- Modularity <-- we are here
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools
Goals of this Lecture

Help you learn:
- How to create high quality modules in C
- The importance of abstraction
- A power programmer knows how to find the abstractions in a large program
- A power programmer knows how to convey a large program's abstractions via its modularity

This is one of the two most important things that will get you promoted from programmer to team leader (,

WHY?

- Help you learn:
  - Arbitrary integer represented by \( σ \)
  - Operation string.

- Operation \( σ \)
- Operation \( Op \)

A List represents a sequence of integers.

- Operation \( insert(p, key) \) inserts \( key \) into \( p \).
- Operation \( nth\_key(p, n) \) returns the \( n \)th key.
- Operation \( len(p) \) returns the length of \( p \).
- Operation \( concat(p, q) \) concatenates \( p \) and \( q \).

Barbara Liskov, a pioneer in CS

"An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type."

Abstract Data Type (ADT)

A data type has a representation

- Abstract Node (int key, Node next)
- Abstract List (Node *first)

and some operations:

- Insert Node (int key, Node next)
- Insert List (List *first)

A power programmer knows how to find the abstractions in a large program.

- Understanding large, complex systems
- How to create high quality modules in C
- A power programmer knows how to convey a large program's abstractions via its modularity

Summary

- A dumb (but correct) implementation
- A smart (but correct) implementation

A smarter implementation

Underspecified behavior

Representation vs. abstraction

ADT modules in C (wrong!)

ADT modules in C (right!)

ADT modules in C (alternate implementation)
What happens compiling client.c

```
#include "list.h"

int f(void) {
    struct List *p, *q;
    p = new();
    q = new();
    insert (p, 6);
    insert (p, 7);
    insert (q, 5);
    concat (p, q);
    concat (q, p);
    return nth_key(q, 1);
}
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
client.c
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
#include "list.h"
```

### enforcement

The moral of the fable is that:

```
Type structure is a syntactic discipline for enforcing levels of abstraction.
```

```
list.h
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
client.c
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
#include "list.h"
```

```
```

### discipline

The moral of the fable is that:

```
Type structure is a syntactic discipline for enforcing levels of abstraction.
```

```
list.h
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
client.c
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
#include "list.h"
```

```
```

### Cheatin’ client

```
list.h
```

```
struct List;
struct List * new(void);
void insert (struct List *p, int key);
void concat (struct List *p, struct List *q);
int nth_key (struct List *p, int n);
```

```
client.c
```

```
```

### Finishing up the module interface

What’s missing?

Well, that depends on your top-down program design. What does the client need? (Can’t tell hasn’t shown you the client)

But probably you’ll want a way to free a List:

```
void free_list(struct list *p);
```
Module Design Principles

We propose 7 module design principles

And illustrate them with 4 examples
• List, string, stdin, SymTable

Continued in next lecture . . .