A Fable
(by John C. Reynolds, 1983)

Once upon a time, there was a university with a peculiar tenure policy. All faculty were tenured and could only be dismissed for moral turpitude: making a false statement in class. Needless to say, the university did not teach computer science. However, it had a renowned department of mathematics.

One semester, there was such a large enrollment in complex variables that two sections were scheduled. In one section, a Professor Descartes announced that a complex number was an ordered pair of reals, and that two complex numbers were equal when their corresponding components were equal. He went on to explain how to convert reals into complex numbers, what “i” was, how to add, multiply, and conjugate complex numbers, and how to find their magnitude.

In the other section, Professor Bessel announced that a complex number was an ordered pair of reals, the first of which was nonnegative, and that two complex numbers were equal if their first components were equal and either the first components were zero or the second components differed by a multiple of 2π. He then told an entirely different story about converting reals, “i”, addition, multiplication, conjugation and magnitude.

Then, after their first classes, an unfortunate mistake in the registrar’s office caused the two sections to be interchanged. Despite this, neither Descartes nor Bessel ever committed moral turpitude, even though each was judged by the other’s definitions. The reason was that they both had an intuitive understanding of type. Having defined complex numbers and the primitive operations upon them, thereafter they spoke at a level of abstraction that encompassed both of their definitions.

The moral of this fable is that:

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

For instance, when Descartes introduced the complex plane, this discipline prevented him from saying Complex = Real = Real, which would have contradicted Bessel’s definition. Instead, he defined the mapping 1: Real = Real → Complex such that 1: x -> x, and proved that this mapping was a bijection.

... More precisely, there is no such thing as the set of complex numbers. Instead, the type “Complex” denotes an abstraction that can be realized or represented by a variety of sets. …

John C. Reynolds
Types, abstraction, and parameters. Polymorphism.

Retelling the Fable

Once upon a time, two software engineering teams were building a library catalog system. In one team, the team leader Dr. Donders announced that a symbol table was a linked list of pairs.

He then went on to define “put” and “get” operations on symbol tables.

```c
int SymTable_put(SymTable_T oSymTable, const char *pcKey, const void *pvValue);
void *SymTable_get(SymTable_T oSymTable, const char *pcKey);
```

In the other team, Dr. Petras announced that a symbol table was an array of linked lists, indexed by a “hash” value.

He then told an entirely different story about “put” and “get.”

Then, after their first team meetings, an IPO caused the two teams to exchange leaders. Each team built a library catalog system using symbol tables with “add” and “lookup,” even though each team was using the other team’s implementation of symbol tables. The reason was that Dr. Donders and Dr. Petras respected the discipline of abstract data types: access the symbol table only through its operations “put” and “get.”

“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

Finally, the team that was using the linked list implementation realized that their performance was slow on large datasets O(N^2) time. They simply substituted the hash-table implementation, and (other than that) not a single line of code had to be changed.
Goals of this Lecture

Help you learn:
- How to create high quality modules in C

Why?
- Abstraction is a powerful (the only?) technique available for understanding large, complex systems
- A software engineer knows how to find the abstractions in a large program
- A software engineer 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 ( . . . to CTO) (what’s the other thing? Hint: it’s on the other side of Washington Road)

Abstract Data Type (ADT)

A data type has a representation

```
struct Node {
    int key;
    struct Node *next;
};
struct List {
    struct Node *first;
};
```

and some operations:

```
struct List * new(void) {
    struct List *p;
    p=(struct List *)malloc (sizeof *p);
    assert (p!=NULL);
    p->first = NULL;
    return p;
}
void insert (struct list *p, int key) {
    struct Node *n;
    n = (struct Node *)malloc(sizeof *n);
    assert (n!=NULL);
    n->key=key; n->next=p->first; p->first=n;
}
void concat (struct list *p, struct list *q) {
    int nth_key (struct list *p, int n);
}
```

If you can’t see the representation (or the implementations of insert, concat, nth_key), then how are you supposed to know what they do?

```
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);
```

Reasoning about client code

A list $p$ represents a sequence of integers $\sigma$. Operation $\text{new}()$ returns a list $p$ representing the empty sequence.

Operation $\text{insert}(p, i)$, if $p$ represents $\sigma$, causes $p$ to now represent $\sigma i$. Operation $\text{concat}(p, q)$, if $p$ represents $\sigma_i$ and $q$ represents $\sigma_j$, causes $p$ to represent $\sigma_i \sigma_j$ and leaves $q$ representing the empty string.

Operation $\text{nth_key}(p, n)$, if $p$ represents $\sigma_i$, returns $\sigma_i$, where the length of $\sigma_i$ is $n$; otherwise, if the length of the string represented by $p$ is $\leq 0$, it returns an arbitrary integer.

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."


A dumb (but correct) implementation

```
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, q);
    return nth_key(q, 1);
}
```
struct Node { int key; struct Node *next;};

struct List { struct Node *first;};

struct List * new(void) {
    struct List *p = (struct List *)malloc(sizeof(*p));
    p->first = NULL;
    return p;
}

void insert (struct List *p, int key) {
    struct Node *n;
    n = (struct Node *)malloc(sizeof *n);
    assert (n!=NULL);
    n->key = key; n->next = p->first; p->first = n;
}

void concat (struct List *p, struct List *q) {
    struct Node *t = p->first;
    if (t==NULL) {
        p->first = q->first;
    } else {
        while (t->next != NULL)
            t = t->next;
        t->next = q->first;
    }
    q->first = NULL;
}

int nth_key (struct List *p, int n) {
    struct Node *t = p->first;
    while (n>0 && t!=NULL) {n--; t=t->next;}
    if (t==NULL) return 6;
    else return t->key;
}

/* No matter which implementation is used, the client program works the same. */
(Might be faster with the smart implementation)

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);    /* 6 */
}

#include "list.h"

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

#include "list.h"

struct List { int len; int *data};

struct List * new(void) {
    struct List *p = (struct List *)malloc(sizeof(*p));
    p->len=0;
    p->data=NULL;
    return p;
}

void insert (struct List *p, int key) {...}

void concat (struct List *p, *q) { ... }

int nth_key (struct List *p, int n) { ... }

#include "list.h"

struct List { int key; struct Node *next;};

struct List * new(void) {
    struct List *p = (struct List *)malloc(sizeof(*p));
    p->first = NULL;
    return p;
}

void insert (struct List *p, int key) {
    struct Node *n;
    n = (struct Node *)malloc(sizeof *n);
    assert (n!=NULL);
    n->key = key; n->next = p->first; p->first = n;
}

void concat (struct List *p, struct List *q) {
    struct Node *t = p->first;
    if (t==NULL) {
        p->first = q->first;
    } else {
        while (t->next != NULL)
            t = t->next;
        t->next = q->first;
    }
    q->first = NULL;
}

int nth_key (struct List *p, int n) {
    struct Node *t = p->first;
    while (n>0 && t!=NULL) {n--; t=t->next;}
    if (t==NULL) return 6;
    else return t->key;
}

#include "list.h"

struct List { int key; struct Node *next;};

struct List * new(void) {
    struct List *p = (struct List *)malloc(sizeof(*p));
    p->first = NULL;
    return p;
}

void insert (struct List *p, int key) {
    struct Node *n;
    n = (struct Node *)malloc(sizeof *n);
    assert (n!=NULL);
    n->key = key; n->next = p->first; p->first = n;
}

void concat (struct List *p, struct List *q) {
    struct Node *t = p->first;
    if (t==NULL) {
        p->first = q->first;
    } else {
        while (t->next != NULL)
            t = t->next;
        t->next = q->first;
    }
    q->first = NULL;
}

int nth_key (struct List *p, int n) {
    struct Node *t = p->first;
    while (n>0 && t!=NULL) {n--; t=t->next;}
    if (t==NULL) return 6;
    else return t->key;
}
null
We propose 7 module design principles

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

Continued in next lecture . . .