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 or making a false statement in class. Needless to say, the university did not do a very good job of enforcing tenure. However, its renowned department of mathematics

One semester, there was such a large enrollment in complex numbers that two sections were scheduled. In one section, Professor Descartes announced that a complex number was an ordered pair of reals, the first of which was imaginary, 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.

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 Blaise 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 types. Having defined complex numbers and the primitive operations upon them, they both 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 Blaise's definition. Instead, he defined the mapping R × R → Complex such that (x, y) → x + iy, and proved that this mapping is a bijection.

More precisely, there is no such thing as the set of complex numbers. Instead, the type "Complex" denotes a mathematical object that can be realized in various ways, such as the class:

```c
struct Complex {
    double re, im;
};
```

and the C code:

```c
Complex c = { 3.0, 4.0 }; // (3+4i)
```

Retelling the Fable

Once upon a time, two software engineering teams were each building a library catalog system. In one team, the team leader, Dr. Dandridge, 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
void *SymTable_get(SymTable_T oSymTable, const char *pcKey)
{
    // ...implementation...
}
```

In the other team, Dr. Petras announced that a symbol table was an array of hash-tables, indexed by a "hash" value.

We then told an entirely different story about "put" and "get:"

```c
int SymTable_put(SymTable_T oSymTable, const char *pcKey, const void *pvValue)
{
    // ...implementation...
}
```

Finally, the team that was using the linked list implementation realized that their performance was slow, and that their implementation was slow as well, since the hash table implementation in the other team was the one the linked list implementation of symbol tables. The reason was that Dr. Dandridge and Dr. Petras respected the discipline of abstract data types: access the symbol table only through its operations, "put" and "get."
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 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 ( . . . to CTO) (what's the other thing? Hint: It's on the southwest side of Washington Road)?

Abstract Data Type (ADT)

A data type has a representation

```c
struct Node {
    int key;
    struct Node *next;
};
```

and some operations:

```c
struct List {
    struct Node *first;
};
```

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

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

```c
void concat (struct List *p, struct List *q) {
    struct List *r;
    r = new();
    for (int i = 0; i < p->len; i++) {
        insert(r, p->data[i]);
    }
    for (int i = 0; i < q->len; i++) {
        insert(r, q->data[i]);
    }
    p->len += q->len;
    p->data = r->data;
    free(r);
}
```

Reasoning about client code

A list represents a sequence of elements \( \sigma \).

Operation \( \text{new}() \) returns a list \( p \) representing the empty sequence.
Operation \( \text{insert}(\sigma_j, \sigma) \), if \( \sigma_j \) represents \( \sigma \), causes \( p \) now to represent \( \sigma \).

Operation \( \text{concat}(p, q) \), if \( p \) represents \( \sigma \) and \( q \) represents \( \sigma' \), causes \( p \) to represent \( \sigma \sigma' \) and leaves \( q \) representing the empty sequence.

Operation \( \text{nth}_\text{key}(\sigma, i) \), if \( \sigma \) represents \( \sigma_j \) where the length of \( \sigma \) is \( n \), returns \( i \); otherwise (if the length of the string represented by \( \sigma \) is \( n \)), it returns \( i \).

```c
int nth_key (struct List *p, int n) {
    if (0 <= n && n < p->len)
        return p->data[n];
    else return 0;
}
```

A dumb (but correct) implementation

```c
struct List {
    int len;
    int *data;
};
```

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

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

```c
void concat (struct List *p, struct List *q) {
    struct List *r;
    r = new();
    for (int i = 0; i < p->len; i++) {
        insert(r, p->data[i]);
    }
    for (int i = 0; i < q->len; i++) {
        insert(r, q->data[i]);
    }
    p->len += q->len;
    p->data = r->data;
    free(r);
}
```

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 these objects. This means that an abstract data type can be defined by defining the characterizing operations for that type."

A smarter implementation

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

struct List {struct Node *first;};

void insert (struct List *p, int key) {
    struct Node *n = (struct Node *)malloc(sizeof *n);
    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;
}

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

client.c

#include "list.h"

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

Underspecified behavior

No matter which implementation is used, the client program works "the same."

(Might be faster with the smart implementation)

This is OK! Client program is not supposed to rely on unspecified behavior. If it does, then installing a different implementation might cause the client to behave differently, in which case, too bad for the client.

ADT modules in C (wrong!)

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) {
    if (0 <= key && key < p->len)
        p->data[key] = key;
    else if (key < 0)
        p->data[p->len] = key;
    else return;
}

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;
}

int nth_key (struct List *p, int n) {
    struct Node *t = p->first;
    if (0 <= n && n < p->len)
        return p->data[n];
    else if (n >= p->len)
        return nth_key(q,n-p->len);
    else return t->key;
}

client.c

#include "list.h"

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

This is not OK! If n is large, it returns an arbitrary integer.

ADT modules in C (right!)

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) {
    if (0 <= key && key < p->len)
        p->data[key] = key;
    else if (key < 0)
        p->data[p->len] = key;
    else return;
}

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

client.c

#include "list.h"

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

This is OK! Client program is not supposed to rely on unspecified behavior. If it does, then installing a different implementation might cause the client to behave differently, in which case, too bad for the client.

ADT modules in C (alternate implementation)

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) {
    if (0 <= key && key < p->len)
        p->data[key] = key;
    else if (key < 0)
        p->data[p->len] = key;
    else return;
}

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

client.c

#include "list.h"

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

This is not OK! If n is large, it returns an arbitrary integer.
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);
}

Cheatin' client

 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);

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);

struct List

int f(void) {
  struct List *p, *q;
  p = new();
  if (p->len > 0)
    return p->data[0];
  else return 8;
}

Finishing up the module interface

void free_list(struct List *p);

client.c

#include "list.h"

struct List {int len; int *data};

int f(void) {
  struct List *p, *q;
  p = new();
  if (p->len > 0)
    return p->data[0];
  else return 8;
}

freeing a List

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

void free_list(struct List *p) {
  struct Node *u, *t = p->first;free (p);while (t!=NULL) {
    u=t->next;
    free(t);t=u;
  }
}
We propose 7 module design principles
And illustrate them with 4 examples
- List, string, stdio, SymTable
Continued in next lecture . . .