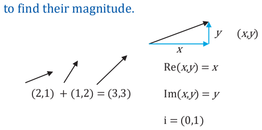
Princeton University
Computer Science 217: Introduction to Programming Systems

Modules and Interfaces

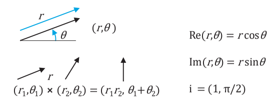
A Fable (by John C. Reynolds, 1983)

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, 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

A Fable

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 f: Real×Real→Complex such that f(x,y)=x+i×y, and proved that this mapping is a bijection.

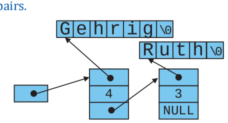
. . .

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

John C. Reynolds.
Types, abstraction, and parametric polymorphism.
Proceedings of the 9th IFIP World Computer Congress, 1983.

Retelling the Fable

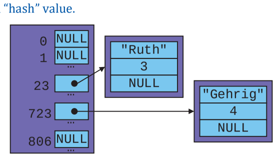
Once upon a time, two software engineering teams were each building a library catalog system. In one team, the team leader Dr. Dondero announced that a symbol table was a linked list of pairs.



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

```
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. Dondero and Dr. Petras respected the *discipline* of abstract data types: access the symbol table only through its *operations,* "put" and "get."

Retelling the Fable

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.

"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

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*

```
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;
}
```

An abstract data type has a *hidden representation;* all "client" code must access the type through its *interface* operations:

```
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);
```


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."

Barbara Liskov and Stephen Zilles. "Programming with Abstract Data Types." *ACM SIGPLAN Conference on Very High Level Languages,* April 1974.

Specifications

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);
```

A List p **represents** a sequence of integers σ.

Operation `new()` returns a list p **representing** the empty sequence.

Operation `insert`(p,i), if p represents σ, causes p to now represent $i\cdot\sigma$.

Operation `concat`(p,q), if p represents σ_1 and q represents σ_2, causes p to represent $\sigma_1\cdot\sigma_2$ and leaves q representing σ_2.

Operation `nth_key`(p,n), if p represents $\sigma_1\cdot i\cdot\sigma_2$ where the length of σ_1 is n, returns i; otherwise (if the length of the string represented by p is $\leq n$), it returns an arbitrary integer.

Reasoning about client code

A List p represents a sequence of integers σ.

Operation `new()` returns a list p representing the empty sequence.

Operation `insert`(p,i), if p represents σ, causes p to now represent $i\cdot\sigma$.

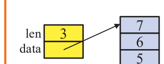
Operation `concat`(p,q), if p represents σ_1 and q represents σ_2, causes p to represent $\sigma_1\cdot\sigma_2$ and leaves q representing the empty string.

Operation `nth_key`(p,n), if p represents $\sigma_1\cdot i\cdot\sigma_2$ where the length of σ_1 is n, returns i; otherwise (if the length of the string represented by p is $\leq n$), it returns an arbitrary integer.

```
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);
```

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

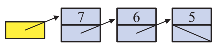

A dumb (but correct) implementation




```
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) {
  int i;
  int *a = (int *)malloc((p->len+1)*sizeof(int));
  for (i=0; i<p->len; i++)
      a[i+1]=p->data[i];
  a[0]=key;
  p->len += 1;
  p->data = a;
}

void concat (struct List *p,
             struct List *q) {
  int i;
  int *a = (int *)malloc((p->len+q->len)*sizeof(int));
  for (i=0; i<p->len; i++)
      a[i]=p->data[i];
  for (i=0; i<q->len; i++)
      a[p->len+i]=q->data[i];
  p->len += q->len;
  p->data = a;
  q->len = 0;
  q->data = NULL;
}

int nth_key (struct List *p, int n) {
  if (0 <= n && n < p->len)
     return p->data[n];
  else return 7;
}
```

A smarter implementation

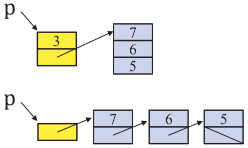
first



```
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;
}
```


Representation vs. abstraction

p



p:[7,6,5]

p:[7,6,5]

```
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);
}
```

```
p:[]
p:[]  q:[]
p:[6] q:[]
p:[7,6]  q:[]
p:[7,6]  q:[5]
p:[7,6,5]  q:[]
p:[]   q:[7,6,5]
return 6
```

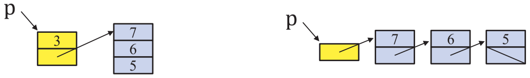
No matter which implementation is used, the client program works "the same."

(Might be faster with the smart implementation)

Underspecified behavior

Operation `nth_key`(p,n), if p represents $\sigma_1 \cdot i \cdot \sigma_2$ where the length of σ_1 is n, returns i;
otherwise (if the length of the string represented by p is $\leq n$), it returns an arbitrary integer.

p



```
int nth_key (struct List *p, int n) {
  if (0 <= n && n < p->len)
    return p->data[n];
  else return 7;
}
```

```
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;
}
```

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

If you put the representation here, then it's not an **abstract** data type, it's just a data type.

(Many C programmers program this way because they don't know any better.)

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);
}
```

list_array.c

```
#include "list.h"

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) { ... }
```


ADT modules in C (right!)

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

```
#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);
}
```

list_array.c

```
#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) { ... }
```


ADT modules in C (alternate implementation)

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

```
#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);
}
```

list_linked.c

```
#include "list.h"

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) {...}

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

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

What happens compiling client.c

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

```
#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);

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);
}
```

Never does any of:

`p->field`

`sizeof (struct List)`

`sizeof (*p)`

enforcement

[John Reynolds]

The moral of this 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);
```

Putting **`struct List;`** here, instead of **`struct List {fields...};`**

enforces the abstraction: it prevents client.c from accessing the fields of the struct.

discipline

[John Reynolds]

The moral of this fable is that:

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

list.h

client.c

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

list_linked.c

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

Arranging your ADTs and their clients in .c files like this, with the interface in .h files, is a *discipline* of programming, to enforce levels of abstraction, that you should use in C programming.

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

```
#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;
}
```

A couple of slides ago, I wrote,
"Putting **`struct List;`** in list.h instead of **`struct List {fields...};`** **enforces** the abstraction: it prevents client.c from accessing the fields of the struct."

Well, the enforcer has its limits. A boneheaded client can always find its way around the enforcement. That leads to brittle, buggy programs!

Finishing up the module interface

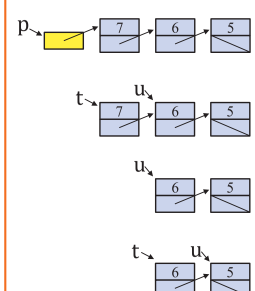
```
struct List;

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);
```

What's missing?
Well, that depends on your top-down program design.
What does the client need? (Can't tell; I haven't shown you the client)

But probably you'll want a way to *free* a List:

```
void free_list(struct list *p);
```


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;
  }
}
```

Module Design Principles

We propose 7 module design principles

And illustrate them with 4 examples

- List, string, stdio, SymTable

Continued in next lecture . . .