Princeton University

Computer Science 217: Introduction to Programming Systems

Modules and Interfaces

Barbara Liskov

Turing Award winner 2008,
"For contributions to practical and theoretical foundations of programming language and system design, especially related to **data abstraction,** fault tolerance, and distributed computing."

COS 217 Midterm

When/where?

- In class, Thursday October 25; rooms to be announced

What?

- C programming, including string and stdio
- Numeric representations and types in C
- Programming in the large: modularity, building, testing, debugging
- Readings, lectures, precepts, assignments, through *this week*
- Mixture of short-answer questions and writing snippets of code

How?

- Closed book, closed notes
- No electronic anything
- Interfaces of relevant functions will be provided

Old exams and study guide will be posted on schedule page

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

Agenda

A good module:

- **Encapsulates data**
- Manages resources
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Encapsulation

A well-designed module encapsulates data

- An interface should hide implementation details
- A module should use its functions to encapsulate its data
- A module should not allow clients to manipulate the data directly

Why?

- **Clarity**: Encourages abstraction
- **Security**: Clients cannot corrupt object by changing its data in unintended ways
- **Flexibility**: Allows implementation to change – even the data structure – without affecting clients

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*:

```
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.

Encapsulation with ADTs (wrong!)

list.h

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

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);
```

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_linked.c

```
#include "list.h"

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) { ... }
```


Encapsulation with ADTs (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);
```

Including only the declaration in header file **enforces** the abstration: it keeps clients from accessing fields of the struct, allowing implementation to change

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) { ... }
```


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.

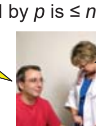
Operation new() returns a list *p* **representing** the empty sequence.

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 ≤ *n*), it returns an arbitrary integer.

This is OK, but not ideal.
Client programs relying on unspecified behavior might break with a new implementation.



Reasoning About Client Code

The *specifications* allow reasoning about the effects of client code.

```
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);
}
```

```
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
```


Object-Oriented Thinking

C is not inherently an object-oriented language, but can use language features to encourage object-oriented thinking

```
typedef struct List *List_T;

List_T new(void);

void insert (List_T p, int key);

void concat (List_T p, List_T q);

int nth_key (List_T p, int n);

void free_list (List_T p);
```

"Opaque" pointer type

- Interface provides `List_T` abbreviation for client
  - Interface encourages client to think of **objects** (not structures) and **object references** (not pointers to structures)
- Client still cannot access data directly; data is "opaque" to the client

iClicker Question

Q: What's the weakest assertion you can make that guarantees the following code won't crash:

```
int a[1000];  int i, c;
assert ( . . . );
c=getchar();  i=0;
while (isalpha(c))
   { a[i++]=c; c=getchar(); }
a[i]='\0';
```

A. assert (strlen(a)<1000)
B. assert (sizeof(stdin)<1000)
C. assert (i<1000);
D. assert (1);
E. assert (0);

Agenda

A good module:

- Encapsulates data
- **Manages resources**
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Resource Management

A well-designed module manages resources consistently

- A module should free a resource if and only if the module has allocated that resource
- Examples
  - Object allocates memory ↔ object frees memory
  - Object opens file ↔ object closes file

Why?

- Allocating and freeing resources at different levels is error-prone
  - Forget to free memory ⇒ memory leak
  - Forget to allocate memory ⇒ dangling pointer, seg fault
  - Forget to close file ⇒ inefficient use of a limited resource
  - Forget to open file ⇒ dangling pointer, seg fault

Resource Management in `stdio`

`fopen()` allocates memory for **`FILE`** struct, obtains file descriptor from OS

`fclose()` frees memory associated with **`FILE`** struct, releases file descriptor back to OS

Resources in Assignment 3

Who allocates and frees the key strings in symbol table?

Reasonable options:

(1) Client allocates and frees strings
- **`SymTable_put()`** does not create copy of given string
- **`SymTable_remove()`** does not free the string
- **`SymTable_free()`** does not free remaining strings

(2) SymTable object allocates and frees strings
- **`SymTable_put()`** creates copy of given string
- **`SymTable_remove()`** frees the string
- **`SymTable_free()`** frees all remaining strings

Our choice: (2)

- With option (1) client could corrupt the SymTable object (as described in last lecture)

Passing Resource Ownership

Violations of expected resource ownership should be noted explicitly in function comments

```
somefile.h

…

void *f(void);
/* …
   This function allocates memory for
   the returned object.  You (the caller)
   own that memory, and so are responsible
   for freeing it when you no longer
   need it. */

…
```

Agenda

A good module:

- Encapsulates data
- Manages resources
- **Is consistent**
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Consistency

A well-designed module is consistent

- A function's name should indicate its module
  - Facilitates maintenance programming
    - Programmer can find functions more quickly
  - Reduces likelihood of name collisions
    - From different programmers, different software vendors, etc.
- A module's functions should use a consistent parameter order
  - Facilitates writing client code

Consistency in `string.h`

string

```
/* string.h */

size_t strlen(const char *s);
char   *strcpy(char *dest, const char *src);
char   *strncpy(char *dest, const char *src, size_t n);
char   *strcat(char *dest, const char *src);
char   *strncat(char *dest, const char *src, size_t n);
int     strcmp(const char *s1, const char *s2);
int     strncmp(const char *s1, const char *s2, size_t n);
char   *strstr(const char *haystack, const char *needle);
void   *memcpy(void *dest, const void *src, size_t n);
int     memcmp(const void *s1, const void *s2, size_t n);
…
```

Are function names consistent?

Is parameter order consistent?