COS 217: Introduction to Programming Systems

Principles for Module Design

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike)
Chapter 4

PRINCETON UNIVERSITY

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
- Modularity is a key manifestation of abstraction
- A mature programmer knows how to identify good abstractions in large programs
- A mature programmer knows how to convey a large program's abstractions via its modularity, including the separation from interface from implementation

Agenda

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources
- Provides a consistent interface
- Provides a minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

We will use the List, String, and
Symbol Table modules as examples

A Good Module Establishes a Clear Contract

Here is what you know about me (e.g., I am a list)

Here are the operations you can ask me to perform

There is nothing else I will do for you, or you can know about me or do to me

- Good for the module providing the interface
  - Constraints clients, so don't have to worry about them messing with internals
- Good for client modules
  - Know what they can and can't do, and needn't worry about messing the module up

Encapsulation + Information Hiding

Encapsulation: bundling together data and methods that operate on the data, restricting access by means other than those methods

- An interface should hide implementation details
- A module should use its functions to encapsulate its data: clients can only access the module through the functions in the 'contract'
- A module should not allow clients to manipulate its internal data directly

Why?

- **Clarity**: Encourages abstraction: clients know what they can and cannot do
- **Security**: Clients cannot corrupt object by changing its data in unintended ways
- **Flexibility**: Allows implementation to change – even the underlying representation, e.g. data structure – without affecting clients

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.

i.e. Client needn't, and doesn't, know the representation

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."

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()
{
  struct List *p;
  p = calloc(1, sizeof(*p));
  assert(p != NULL);
  return p;
}

void insert(struct List *p, int key)
{
  struct Node *n;
  n = 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 insert(struct List *p, int key);
void concat(struct List *p,
            struct List *q);
int nth_key(struct List *p, int n);
...
```

Encapsulation with ADTs (wrong!)

Nothing stops a client from doing this!

```
p->first = NULL;
```

list.h

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

struct List *new();
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.

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()
{
  struct List *p;
  p = calloc(1, sizeof(*p));
  assert(p != NULL);
  return p;
}

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

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

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

Encapsulation with ADTs (correct!)

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

list.h

```c
struct List;


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

Even compiler doesn't know implementation when compiling client.c. So will give error at "p->first = NULL;"

client.c

```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

```c
#include "list.h"

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

struct List *new()
{
  struct List *p;
  p = calloc(1, sizeof(*p));
  assert(p != NULL);
  return p;
}

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

Discussed in Precept

”Object-orientedness” with `typedefs`

Extensibility with function pointers

Question from Precept: Abstract Data Type?

Q: Is a string, as used by the `<string.h>` module an ADT?

A. Yes – clients can't know the implementation of `strcpy`, etc.

B. Yes – clients can't know the representation of strings.

C. No – clients can know the implementation of `strcpy`, etc.

D. No – clients can know the representation of strings.

E. No – strings are not a datatype.

D

We know the underlying representation of strings.

Clients can manipulate the string's state directly, not through the interface.

Module Specification

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

Specification:

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.

Ugh!? But specification makes clear that this is now the responsibility of the client. Note: already true for arrays in C.

Specifications Establish a Contract

A well-designed module establishes a contract

- A module should establish a contract with its clients
- The contract should describe what each function does, especially:
  - Meanings of parameters
  - Work performed
  - Meaning of return value
  - Side effects

Why?

- Facilitates cooperation between multiple programmers
- Assigns <u>blame</u> to contract violators!
  - If all *your* functions have precise contracts and implement them correctly, then the bug must be in *someone else's* code

How? Comments in module interface

Specifications Allow Reasoning About Code

Can trace behavior of client code without access to or knowing about how the list ADT is implemented. And even if it's not yet implemented (by another team, say)

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

Agenda

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources
- Provides a consistent interface
- Provides a minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

Resource Management

A well-designed module manages resources consistently

- A module should release a resource *iff* the module has claimed that resource
  - Allocate and free memory, open and close file, etc.
- Should client allocate and free a resource, or should module?
  - Recall Symbol Table owning the key string from prior lecture
- Can be reasons for both, but generally who allocates should free

Why?

- Claiming and releasing 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

Exceptions to this rule should be clearly documented in comments

Passing Resource Ownership

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

```
/* somefile.h */

...

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

…
```

e.g., `strdup` in the string module, and an example in A4

Agenda

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources appropriately
- Provides a consistent and minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

Consistency

A function's **name** should indicate its **module,** e.g. SymTable_put()

- Facilitates maintenance programming
  - Programmer can find functions more quickly
- Reduces likelihood of name collisions (so important for non-static functions)
  - From different programmers, different software vendors, etc.

A module's functions should use a consistent parameter order

- Facilitates writing client code

How Does C Do: Consistency in `string.h`

Are function names consistent?

```
/* 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);
...
```

Is parameter order consistent?

Consistency in `symtable.h`

Are function names consistent?

```
SymTable_T  SymTable_new(void);
void        SymTable_free(SymTable_T oSymTable);
size_t      SymTable_getLength(SymTable_T oSymTable);
int         SymTable_put(SymTable_T oSymTable, const char *pcKey, const void *pvValue);
void       *SymTable_replace(SymTable_T oSymTable, const char *pcKey, const void *pvValue);
int         SymTable_contains(SymTable_T oSymTable, const char *pcKey);
void       *SymTable_get(SymTable_T oSymTable, const char *pcKey);
void       *SymTable_remove(SymTable_T oSymTable, const char *pcKey);
void        SymTable_map(SymTable_T oSymTable,
                         void (*pfApply)(const char *pcKey, void *pvValue, void *pvExtra),
                         const void *pvExtra);
```

Is parameter order consistent?

Let’s make List comply ...

List

- (–) Every function name doesn't begin with “List_”
- (+) First parameter identifies List_T object

```
typedef struct List *List_T;

List_T List_new();

void List_insert(List_T p, int key);

void List_concat(List_T p, List_T q);

int  List_nth_key(List_T p, int n);

void List_free(List_T p);
```

Oops, let’s fix that!

List (revised)

- (+) Each function name begins with “List_”
- (+) First parameter identifies List_T object

Minimalism

A well-designed module has a minimal interface

- Function declaration should be in a module's interface if and only if:
  - The function is necessary for functionality, or
  - The function is necessary for clarity of client code

Why?

- More functions $\Rightarrow$ higher learning costs, higher maintenance costs

Minimalism: Do we Need SymTable_contains?

Q: Assignment 3's interface has both `SymTable_get()` (which returns NULL if the key is not found) and `SymTable_contains()` – is the latter necessary?

A. No – should be eliminated
B. Yes – necessary for functionality
C. Yes – necessary for efficiency
D. Yes – necessary for clarity

B

SymTable bindings can have NULL values, but SymTable_get() can't tell these apart from keys that aren't in the table.

Now Hash This One Out

Q: Assignment 3 has `SymTable_hash()` defined in `symtablehash.c`'s implementation, but not the `symtable.h` interface. Is this good design?

A. No – should be in interface to enable functionality

B. No – should be in interface to enable clarity

C. Yes – should remain an implementation detail

C

It is only ever used internally, and only in a hash table implementation.

See discussion of "static" functions in precept

Agenda

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources appropriately
- Provides a consistent and minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

Error Handling

A well-designed module should:

- **Detect** errors (in C, with if statements or asserts)
- **Handle and/or report** errors.

Java provides language features to throw and catch exceptions

In C, some (less elegant) ways include:

- Use a function return value, that client can check and interpret
- Use a (call-by-reference) function parameter, that client can check and interpret
- Use a global variable, that client can check and interpret
- Use an assert, and terminate the execution
- (Languages like Java provide features to throw exceptions and catch them)

We recommend different approaches for user, programming errors

Handling Errors in C

C options for **detecting** errors

- `if` statement
- `assert` macro

C options for **handling** errors

- Write message to `stderr`
  - Impossible in many embedded applications
- Recover and proceed
  - Sometimes impossible
- Abort process
  - Often undesirable

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Use **function return value**?

```
int div(int dividend, int divisor, int *quotient)
{
   if (divisor == 0)
      return 0;
   ...
   *quotient = dividend / divisor;
   return 1;
}
...
successful = div(5, 3, &quo);
if (!successful)
   /* Handle the error */
```

Awkward if return value has some other natural purpose

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Set **global variable**?

```
int successful;
...
int div(int dividend, int divisor)
{
   if (divisor == 0) {
      successful = 0;
      return 0;
   }
   successful = 1;
   return dividend / divisor;
}
...
quo = div(5, 3);
if (!successful)
   /* Handle the error */
```

  - Easy for client to forget to check
  - Bad for multi-threaded programming
  - Some standard C library functions set `errno` global variable

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Use **call-by-reference parameter**?

```
int div(int dividend, int divisor, int *successful)
{
   if (divisor == 0) {
      *successful = 0;
      return 0;
   }
   *successful = 1;
   return dividend / divisor;
}
...
quo = div(5, 3, &successful);
if (!successful)
   /* Handle the error */
```

Awkward for client; must pass additional argument

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Call `assert` **macro**?

```
int div(int dividend, int divisor)
{
   assert(divisor != 0);
   return dividend / divisor;
}
...
quo = div(5, 3);
```

- Asserts could be disabled
- Error terminates the process!

User Errors

Errors that are best handled by the human user (often made by the human user)

Errors that "could happen"

Example: Bad data in `stdin` or command line input, too much/little data

Recommendation:

- Use `if` statement to detect
- Handle immediately if possible, or…
- Report to client via return value or call-by-reference parameter
  - Don't use global variables (not thread-safe, easy to ignore)

Programmer Errors

Errors best handled by a programmer (often made by the programmer)

Errors that "should never happen"

Examples: pointer parameter is NULL but should not be (violates a module's contract); invariant is true at entry into function but not at exit

For now, use `assert` to detect and handle, as a user can't do anything about it

- Programmer needs to get involved

The distinction sometimes is unclear

- Example: Write to file fails because disk is full
- Example: Divisor argument to `div()` is 0, due to user input

Default: treat as user error

Agenda

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources appropriately
- Provides a consistent and minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

Strong Cohesion and Weak Coupling

Strong cohesion

- A module's functions should be strongly related to each other

Weak coupling

- Module should be weakly connected to other modules
- Interaction **within** modules should be more intense than **among** modules

Why?

- Strong cohesion facilitates abstraction, keeps modules small enough
- Weak coupling makes program easier to modify and modules easier to reuse

Strong Cohesion Examples

List

(+) All functions are related to the encapsulated data

string.h

(+) Most functions are related to string handling
(-) Some functions are not related to string handling: `memcpy`, `memcmp`...
(+) But those functions are similar to string-handling functions

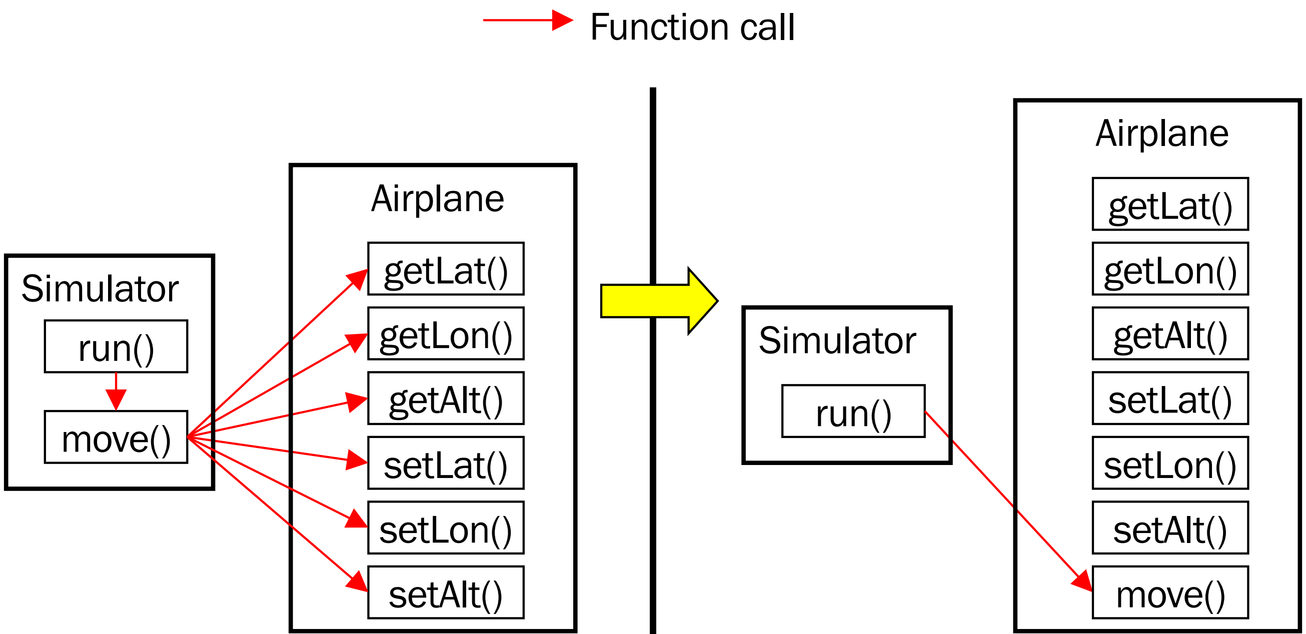
stdio.h

(+) Most functions are related to I/O
(-) Some functions don't do I/O: `sprintf`, `sscanf`
(+) But those functions are similar to I/O functions

SymTable

(+) All functions are related to the encapsulated data

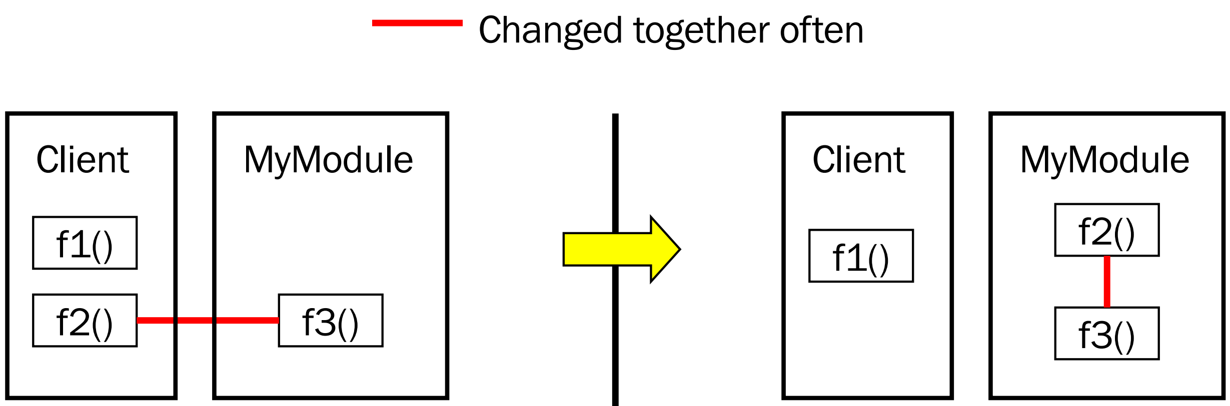
Design-time Weak Coupling Example



- Simulator module calls many functions in Airplane
- Strong design-time coupling

- Simulator module calls few functions in Airplane
- Weak design-time coupling

Maintenance-time Weak Coupling Example



- Maintenance programmer changes Client and MyModule together frequently
- Strong maintenance-time coupling

- Maintenance programmer changes Client and MyModule together infrequently
- Weak maintenance-time coupling

Achieving Weak Coupling

Achieving weak coupling could involve **refactoring** code:

- Move code from client to module (shown)
- Move code from module to client (not shown)
- Move code from client and module to a new module (not shown)

Summary

@danist07

A good module:

- Provides encapsulation and establishes a contract
- Manages resources
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Has strong cohesion and weak coupling

Sample Exam Questions

S17 Exam2 Q6c: What changes would be needed in a callback function for your A3 symbol table's map function if the implementation of the symbol table is changed from using a linked list to using a hash table?

Sample Exam Question (Spring 2020 Exam 2)

Consider the following program, which consists of 6 files:
{ a.h, a.c, b.h, b.c, c.h, c.c}.

a.h:

```
#include <stddef.h>
/* struct a is a thing. you can't see inside, though.
better yet, just think of it as an object */
typedef struct a * a_T;
a_T a_new(const char* src);
size_t aT_to_size_t(a_T a);
void a_free(a_T a);
```

a.c:

```
#include <stdlib.h>
#include <string.h>
#include "a.h"
struct a { size_t a;};
a_T a_new(const char* src) {
 char* res = strstr(src, "a");
 a_T a = malloc(sizeof(*a));
 if(res == NULL) a->a = 0;
 else a->a = res-src;
 return a;
}
size_t aT_to_size_t(a_T a) {
 return a->a;
}
void a_free(a_T a) {
 free(a);
}
```

b.h:

```
/* I need a.h to know what an a_T is. */
#include "a.h"
a_T getAnA(void);
```

b.c:

```
#include "b.h"
#include <stdio.h>enum { LIMIT = 100 };
a_T getAnA(void) {
 char buf[LIMIT];
 scanf("%s", buf);
 return a_new(buf);
}
```

c.h:

```
#include <stdio.h>
#include "b.h"
```

c.c:

```
#include "c.h"
int main(void) {
 a_T at = getAnA();
 printf("%lu\n", aT_to_size_t(at));
 return 0;
}
```

What ambiguity or potential problem is exposed to clients of module A via the return value of the `a_new` function?

Hint – consider the following inputs to the client program :
`ensign`, `lieutenant`, `commander`, `captain`, `admiral`.

Sample Exam Question (Fall 2015, Exam 2)

Consider the **`Queue`** interface:

```
/* A Queue is a first-in-first-out data structure.*/
/* First node of the queue*/
struct QueueNode * first;
/* The last node of the queue*/
struct QueueNode * last;
/* The number of elements in the queue */
int count;
/* Initialize the Queue */
void Queue_init ( void );
/* Free the resources consumed by the Queue */
void Queue_free ( void );
/* Return the number of items in the Queue */
int Queue_getCount ( void );
/* Add item e to the end of the Queue. Return 1 (TRUE) if successful and 0 (FALSE) if memory is exhausted. */
int Queue_enqueue ( void * e );
/* Remove the item at the front of the queue and return it. */
void * Queue_dequeue ( void );
```

Q8b: Briefly describe *two* design problems with this code (i.e., two ways the .h file violates standard practice for modular software development) and how they should be fixed?