



Modularity

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

1



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

2

Module Design Heuristics



- We propose 7 module design heuristics
- Let's consider one at a time...

3

Interfaces



(1) A well-designed module separates interface and implementation

- Why?
 - Hides implementation details from clients
 - Thus facilitating abstraction
 - Allows separate compilation of each implementation
 - Thus allowing partial builds

4

Interface Example 1



- Stack: A stack whose items are strings
 - Data structure
 - Linked list
 - Algorithms
 - **new**: Create a new Stack object and return it (or NULL if not enough memory)
 - **free**: Free the given Stack object
 - **push**: Push the given string onto the given Stack object and return 1 (or 0 if not enough memory)
 - **top**: Return the top item of the given Stack object
 - **pop**: Pop a string from the given Stack object and discard it
 - **isEmpty**: Return 1 the given Stack object is empty, 0 otherwise

5

Interfaces Example 1



- Stack (version 1)

```
/* stack.c */
struct Node {
    const char *item;
    struct Node *next;
};
struct Stack {
    struct Node *first;
};

struct Stack *Stack_new(void) {...}
void Stack_free(struct Stack *s) {...}
int Stack_push(struct Stack *s, const char *item) {...}
char *Stack_top(struct Stack *s) {...}
void Stack_pop(struct Stack *s) {...}
int Stack_isEmpty(struct Stack *s) {...}

/* client.c */
#include "stack.c"

/* Use the functions
defined in stack.c. */
```

- Stack module consists of one file (stack.c); no interface
- Problem: Change stack.c => must rebuild stack.c **and client**
- Problem: Client “sees” Stack function definitions; poor abstraction

6

Interfaces Example 1



- Stack (version 2)

```
/* stack.h */

struct Node {
    const char *item;
    struct Node *next;
};
struct Stack {
    struct Node *first;
};

struct Stack *Stack_new(void);
void Stack_free(struct Stack *s);
int Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void Stack_pop(struct Stack *s);
int Stack_isEmpty(struct Stack *s);
```

- Stack module consists of two files:
 - (1) stack.h (the interface) declares functions and defines data structures

7

Interfaces Example 1



- Stack (version 2)

```
/* stack.c */

#include "stack.h"

struct Stack *Stack_new(void) {...}
void Stack_free(struct Stack *s) {...}
int Stack_push(struct Stack *s, const char *item) {...}
char *Stack_top(struct Stack *s) {...}
void Stack_pop(struct Stack *s) {...}
int Stack_isEmpty(struct Stack *s) {...}
```

- (2) stack.c (the implementation) defines functions
 - #includes stack.h so
 - Compiler can check consistency of function declarations and definitions
 - Functions have access to data structures

8

Interfaces Example 1



- Stack (version 2)

```
/* client.c */  
  
#include "stack.h"  
  
/* Use the functions declared in stack.h. */
```

- Client #includes only the interface
- Change stack.c => must rebuild stack.c, **but not the client**
- Client does not “see” Stack function definitions; better abstraction

9

Interface Example 2



- string (also recall Str from Assignment 2)

```
/* 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);  
char *strcmp(const char *s, const char *t);  
char *strncmp(const char *s, const char *t, size_t n);  
char *strstr(const char *haystack, const char *needle);  
...
```

10

Interface Example 3



- stdio (from C90, vastly simplified)

```
/* stdio.h */

struct FILE {
    int cnt; /* characters left */
    char *ptr; /* next character position */
    char *base; /* location of buffer */
    int flag; /* mode of file access */
    int fd; /* file descriptor */
};

#define OPEN_MAX 20
FILE _iob[OPEN_MAX];

#define stdin (&_iob[0]);
#define stdout (&_iob[1]);
#define stderr (&_iob[2]);
...
```

Don't be concerned
with details

11

Interface Example 3



- stdio (cont.)

```
...
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *f);
int fflush(FILE *f);

int fgetc(FILE *f);
int getc(FILE *f);
int getchar(void);

int putc(int c, FILE *f);
int putchar(int c);

int fscanf(FILE *f, const char *format, ...);
int scanf(const char *format, ...);

int fprintf(FILE *f, const char *format, ...);
int printf(const char *format, ...);
...
```

12

Encapsulation



(2) 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

13

Encapsulation Example 1



• Stack (version 1)

```
/* stack.h */  
  
struct Node {  
    const char *item;  
    struct Node *next;  
};  
struct Stack {  
    struct Node *first;  
};  
  
struct Stack *Stack_new(void);  
void Stack_free(struct Stack *s);  
void Stack_push(struct Stack *s, const char *item);  
char *Stack_top(struct Stack *s);  
void Stack_pop(struct Stack *s);  
int Stack_isEmpty(struct Stack *s);
```

Structure type definitions
in .h file

- That's bad
- Interface reveals how Stack object is implemented (e.g., as a linked list)
- Client can access/change data directly; could corrupt object

14

Encapsulation Example 1



- Stack (version 2)

```
/* stack.h */  
  
struct Stack;  
  
struct Stack *Stack_new(void);  
void Stack_free(struct Stack *s);  
void Stack_push(struct Stack *s, const char *item);  
char *Stack_top(struct Stack *s);  
void Stack_pop(struct Stack *s);  
int Stack_isEmpty(struct Stack *s);
```

Move definition of struct Node to implementation; clients need not know about it

Place *declaration* of struct Stack in interface; move *definition* to implementation

- That's better
- Interface does not reveal how Stack object is implemented
- Client cannot access data directly

Encapsulation Example 1



- Stack (version 3)

```
/* stack.h */  
  
typedef struct Stack * Stack_T;  
  
Stack_T Stack_new(void);  
void Stack_free(Stack_T s);  
void Stack_push(Stack_T s, const char *item);  
char *Stack_top(Stack_T s);  
void Stack_pop(Stack_T s);  
int Stack_isEmpty(Stack_T s);
```

Opaque pointer

- That's better still
- Interface provides "Stack_T" abbreviation for client
- Interface encourages client to view a Stack as an object, not as a (pointer to a) structure
- Client still cannot access data directly; data is "opaque" to the client

Encapsulation Example 2



- string
 - “Stateless” module
 - Has no state to encapsulate!

17

Encapsulation Example 3



- `stdio` `/* stdio.h */`

```
struct FILE {  
    int cnt;    /* characters left */  
    char *ptr; /* next character position */  
    char *base; /* location of buffer */  
    int flag;  /* mode of file access */  
    int fd;    /* file descriptor */  
};  
...
```

- Violates the heuristic
- Programmers can access data directly
 - Can corrupt the FILE object
 - Can write non-portable code
- But the functions are well documented, so
 - Few programmers examine `stdio.h`
 - Few programmers are tempted to access the data directly

Structure type
definition in .h file

18

Resources



(3) 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 \Leftrightarrow object frees memory
 - Object opens file \Leftrightarrow object closes file
- Why?
 - Error-prone to allocate and free resources at different levels

What if module allocates memory and nobody frees it?

What if module frees memory that nobody has allocated?

Resources Example 1



• Stack: Who allocates and frees the strings?

- Reasonable options:
 - (1) Client allocates and frees strings
 - `Stack_push()` does not create copy of given string
 - `Stack_pop()` does not free the popped string
 - `Stack_free()` does not free remaining strings
 - (2) Stack object allocates and frees strings
 - `Stack_push()` creates copy of given string
 - `Stack_pop()` frees the popped string
 - `Stack_free()` frees all remaining strings
- Our choice: (1)

Advantages/
Disadvantages?

Resources Examples 2, 3



- **string**
 - Stateless module
 - Has no resources to manage!
- **stdio**
 - `fopen()` allocates memory, uses file descriptor
 - `fclose()` frees memory, releases file descriptor

21

SymTable Aside



- Consider `SymTable` (from Assignment 3)...
- Who allocates and frees the key strings?
 - 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)

Adv/Disadv (recall last lecture?)

22

Passing Resource Ownership



- Passing resource ownership
 - Should note violations of the heuristic 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. */  
...
```

23

Consistency



(4) 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

24

Consistency Examples



- Stack

- (+) Each function name begins with “Stack_”
- (+) First parameter identifies Stack object

- string

- (+) Each function name begins with “str”
- (+) Destination string parameter comes before source string parameter; mimics assignment

25

Consistency Examples (cont.)



- stdio

```
...
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *f);
int fflush(FILE *f);

int fgetc(FILE *f);
int getc(FILE *f);
int getchar(void);

int putc(int c, FILE *f);
int putchar(int c);

int fscanf(FILE *f, const char *format, ...);
int scanf(const char *format, ...);

int fprintf(FILE *f, const char *format, ...);
int printf(const char *format, ...);
...
```

Are function names consistent?

Is parameter order consistent?

26

Minimization



(5) 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** to make objects complete, or
 - The function is **convenient** for many clients
- Why?
 - More functions => higher learning costs, higher maintenance costs

27

Minimization Example 1



• Stack

```
/* stack.h */  
  
typedef struct Stack *Stack_T ;  
  
Stack_T Stack_new(void) ;  
void Stack_free(Stack_T s) ;  
void Stack_push(Stack_T s, const char *item) ;  
char *Stack_top(Stack_T s) ;  
void Stack_pop(Stack_T s) ;  
int Stack_isEmpty(Stack_T s) ;
```

Should any
functions be
eliminated?

28

Minimization Example 1



- Another Stack function?

```
void Stack_clear(Stack_T s);  
• Pops all items from the Stack object
```

Should the Stack ADT
define Stack_clear()?

29

Minimization Example 2



- 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);  
char *strcmp(const char *s, const char *t);  
char *strncmp(const char *s, const char *t, size_t n);  
char *strstr(const char *haystack, const char *needle);  
...
```

Should any
functions be
eliminated?

30

Minimization Example 3



- `stdio`

```
...  
FILE *fopen(const char *filename, const char *mode);  
int  fclose(FILE *f);  
int  fflush(FILE *f);  
  
int  fgetc(FILE *f);  
int  getc(FILE *f);  
int  getchar(void);  
  
int  putc(int c, FILE *f);  
int  putchar(int c);  
  
int  fscanf(FILE *f, const char *format, ...);  
int  scanf(const char *format, ...);  
  
int  fprintf(FILE *f, const char *format, ...);  
int  printf(const char *format, ...);  
...
```

Should any
functions be
eliminated?

31

SymTable Aside



- Consider `SymTable` (from Assignment 3)

- Declares `SymTable_get()` in interface
- Declares `SymTable_contains()` in interface

Should
`SymTable_contains()`
be eliminated?

32

SymTable Aside (cont.)



- Consider `SymTable` (from Assignment 3)

- Defines `SymTable_hash()` in implementation

Should
`SymTable_hash()` be
declared in interface?

- Incidentally: In C any function should be either:
 - **Non-static**, and **declared** in the interface
 - **Static**, and **not declared** in the interface

33

Error Detection/Handling/Reporting



(6) A well-designed module detects and handles/
reports errors

- A module should:
 - **Detect** errors
 - **Handle** errors if it can; otherwise...
 - **Report** errors to its clients
 - A module often cannot assume what error-handling action its clients prefer

34

Detecting and Handling Errors in C



- C options for **detecting** errors
 - `if` statement
 - `assert` macro
- C options for **handling** errors
 - Print message to `stderr`
 - Impossible in many embedded applications
 - Recover and proceed
 - Sometimes impossible
 - Abort process
 - Often undesirable

35

Reporting Errors in C



- C options for **reporting** errors to client
 - Set **global variable**?
 - Easy for client to forget to check
 - Bad for multi-threaded programming
 - Use **function return value**?
 - Awkward if return value has some other natural purpose
 - Use extra **call-by-reference parameter**?
 - Awkward for client; must pass additional parameter
 - Call **`assert` macro**?
 - Terminates the entire program!
- No option is ideal

What additional option does Java provide?

36

User Errors



Our recommendation: Distinguish between...

(1) **User errors**

- Errors made by human user
- Errors that “could happen”

- Example: Bad data in stdin
- Example: Bad value of command-line argument

- Use `if` statement to detect
- Handle immediately if possible, or...
- Report to client via return value or call-by-reference parameter

37

Programmer Errors



(2) **Programmer errors**

- Errors made by a programmer
- Errors that “should never happen”

- Example: `int` parameter should not be negative, but is
- Example: pointer parameter should not be `NULL`, but is

- Use `assert` to detect and handle

- The distinction sometimes is unclear
 - Example: Write to file fails because disk is full

38

Error Handling Example 1



- Stack

```
/* stack.c */
...
int Stack_push(Stack_T s, const char *item) {
    struct Node *p;
    assert(s != NULL);
    p = (struct Node*)malloc(sizeof(struct Node));
    if (p == NULL) return 0;
    p->item = item;
    p->next = s->first;
    s->first = p;
    return 1;
}
```

- Invalid parameter is **programmer** error
 - Should never happen
 - Detect and handle via **assert**
- Memory allocation failure is **user** error
 - Could happen (huge data set and/or small computer)
 - Detect via **if**; report to client via return value

39

Error Handling Examples 2, 3



- **string**
 - No error detection or handling/reporting
 - Example: NULL parameter to **strlen()** => probable seg fault
- **stdlib**
 - Detects bad input
 - Uses function return values to report failure
 - Note awkwardness of **scanf()**
 - Sets global variable **errno** to indicate reason for failure

40

Establishing Contracts



(7) A well-designed module establishes contracts

- A module should establish contracts with its clients
- Contracts should describe what each function does, esp:
 - Meanings of parameters
 - Work performed
 - Meaning of return value
 - Side effects
- Why?
 - Facilitates cooperation between multiple programmers
 - Assigns blame to contract violators!!!
 - If your functions have precise contracts and implement them correctly, then the bug must be in someone else's code!!!

41

Establishing Contracts in C



- Our recommendation...
- In C, establish contracts via comments in module interface

42

Establishing Contracts Example



- Stack

```
/* stack.h */
...
int Stack_push(Stack_T s, const char *item);
/* Push item onto s. Return 1 (TRUE)
   if successful, or 0 (FALSE) if
   insufficient memory is available. */
...
```

- Comment defines contract:
 - Meaning of function's parameters
 - s is the stack to be affected; item is the item to be pushed
 - Work performed
 - Push item onto s
 - Meaning of return value
 - Indicates success/failure
 - Side effects
 - (None, by default)

43

Summary



- A well-designed module:
 - (1) Separates interface and implementation
 - (2) Encapsulates data
 - (3) Manages resources consistently
 - (4) Is consistent
 - (5) Has a minimal interface
 - (6) Detects and handles/reports errors
 - (7) Establishes contracts

44

Appendix



Two additional heuristics
which are more advanced in nature...

45

Strong Cohesion



(8) A well-designed module has strong cohesion

- A module's functions should be strongly related to each other
- Why?
 - Strong cohesion facilitates abstraction

46

Strong Cohesion Examples



- **Stack**
 - (+) All functions are related to the encapsulated data
- **string**
 - (+) Most functions are related to string handling
 - (-) Some functions are not related to string handling
`memcpy()`, `memmove()`, `memcmp()`, `memchr()`, `memset()`
 - (+) But those functions are similar to string-handling functions
- **stdio**
 - (+) 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

47

Weak Coupling



- (9) **A well-designed module has weak coupling**
 - Module should be weakly connected to other modules in program
 - Interaction **within** modules should be more intense than interaction **among** modules
- **Why? Theoretical observations**
 - Maintenance: Weak coupling makes program easier to modify
 - Reuse: Weak coupling facilitates reuse of modules
- **Why? Empirical evidence**
 - Empirically, modules that are weakly coupled have fewer bugs

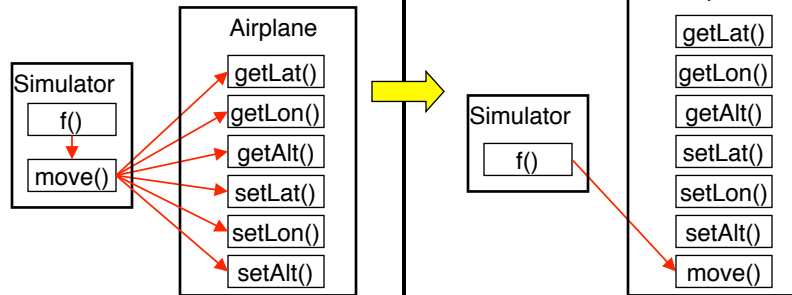
48

Weak Coupling Examples



• Design-time coupling

→ Function call



- Client module calls **many** functions in my module
- **Strong** design-time coupling

- Client module calls **few** functions in my module
- **Weak** design-time coupling

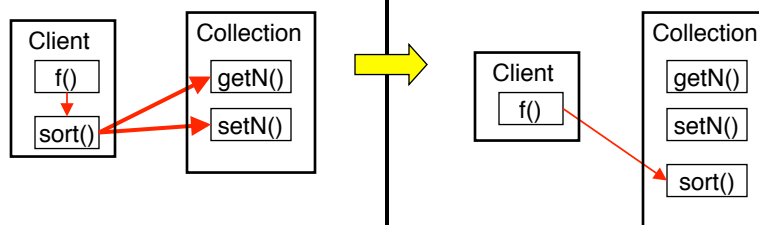
49

Weak Coupling Examples (cont.)



• Run-time coupling

→ Many function calls → One function call



- Client module makes **many** calls to my module
- **Strong** run-time coupling

- Client module makes **few** calls to my module
- **Weak** run-time coupling

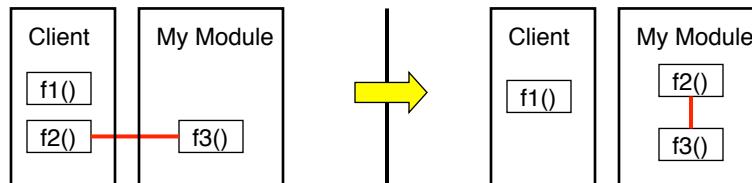
50

Weak Coupling Examples (cont.)



- Maintenance-time coupling

— Changed together often



- Maintenance programmer changes client and my module together frequently
- Strong maintenance-time coupling

- Maintenance programmer changes client and my module together infrequently
- Weak maintenance-time coupling

51

Achieving Weak Coupling



- Achieving weak coupling could involve moving code:

- From clients to my module (shown)
- From my module to clients (not shown)
- From clients and my module to a new module (not shown)

52

Summary



- A well-designed module:
 - (1) Separates interface and implementation
 - (2) Encapsulates data
 - (3) Manages resources consistently
 - (4) Is consistent
 - (5) Has a minimal interface
 - (6) Detects and handles/reports errors
 - (7) Establishes contracts
 - (8) Has strong cohesion**
 - (9) Has weak coupling**