When/where?
• In lecture, W 3/11; Friend 101 (P0{1,2,3,5,6}) and CS 105 (P04)

What?
• C programming, including string and stdio features we’ve seen
• Numeric representations corresponding to C types we’ve seen
• Programming in the large: modularity, building, testing, debugging
• Readings, lectures, precepts: through this week
• Assignments 0-3.

How?
• Mixture of T/F, MC, short-answer, and code snippet writing
• Closed book and notes
• No electronic anything
• Interfaces of relevant functions will be provided

Old exams are posted on schedule page
Modules and Interfaces

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

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

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

and some operations:

```c
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:

```c
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);
```
"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."


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.”
Encapsulation with ADTs (wrong!)

list.h

```c
struct List {int len; int* 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);
```

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

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 List * new(void) {
    struct List *p = (struct List *)malloc(sizeof(*p));
    p->len = 0; p->list=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**

```c
#include "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);
```

**list_linked.c**

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

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

A List $p$ represents a sequence of integers $\sigma$.

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

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

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

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

This is OK! Client programs relying on unspecified behavior might break with a new implementation.

```c
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);
```
List of specifications allows for reasoning about the effects of client code.

```c
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);
}
```

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

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

A. Yes – clients can’t see the implementation of `strcpy`, etc.
B. Yes – clients can’t see the representation of strings.
C. No – clients can see the implementation of `strcpy`, etc.
D. No – clients can see the representation of strings.
E. No – strings are not a datatype.
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)
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
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

```c
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. */
...
```
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)
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`

Are function names consistent?

```c
/* 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);
...
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)
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
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
Q: Assignment 3 has `SymTable_hash()` defined in implementation, but not 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
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)
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
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
C options for **reporting** errors to client (calling function)

- Set **global** variable?

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

- Use **function return value**?

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

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

```c
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
C options for **reporting** errors to client (calling function)
- Call `assert` macro?

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

- Asserts could be disabled
- Error terminates the process!
C options for **reporting** errors to client (calling function)

- No option is ideal

What option does Java provide?
User Errors

Our recommendation: Distinguish between...

(1) **User errors**
- Errors made by human user
- Errors that “could happen”
  - Example: Bad data in `stdin`
  - Example: Too much 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
  - Don’t use global variable
(2) **Programmer errors**

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

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

- For now, use `assert` to detect and handle
  - More info later in the course

**The distinction sometimes is unclear**

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

**Default: user error**
add assert(p) in each of the functions.... try to protect against bad clients

void List_insert (List_T p, int key) {
    assert(p);
    
    
    
}
typedef struct List *List_T;
List_T List_new(void);
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);

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

• This error-handling in List_nth_key is a bit lame.
• How to fix it? Some choices:
  • int List_nth_key (List_T p, int n, int *error);
  • Or, perhaps better: add an interface function,
    int List_length (List_T p); and then,
    Operation nth_key(p, n ), if p represents $\sigma_1 \cdot i \cdot \sigma_2$ where the length of $\sigma_1$ is n, returns $i$; otherwise (if the length of the string represented by p is ≤ n), it fails with an assertion failure or abort( ).
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)
Establishing Contracts

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!!!

How?
- Comments in module interface
Contracts in List

Comment defines contract:
• Meaning of function’s parameters
  • \( p \) is the list to be operated on; \( n \) is the index of an element
• Obligations of caller
  • make sure \( n \) is in range; (implicit) make sure \( p \) is a valid list
• Work performed
  • Return the \( n \)’th element.
• Meaning of return value
• Side effects (none, by default)

/* list.h */

/* Return the \( n \)’th element of the list \( p \), if it exists. Otherwise (if \( n \) is negative or \( \geq \) the length of the list), abort the program. */

int List_nth_key (List_T p, int n);
Comment defines contract:

- Meaning of function’s parameters
  - \( p \) is the list to be queried; \( n \) is the index of an element; \texttt{success} is an error flag
- Obligations of caller
  - (implicit) make sure \( p \) is a valid List
- Work performed
  - Return the \( n \)’th element; set \texttt{success} appropriately
- Meaning of return value
- Side effects: set \texttt{success}
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)
**Strong Cohesion**

A well-designed module has **strong cohesion**
- A module's functions should be strongly related to each other

**Why?**
- Strong cohesion facilitates abstraction
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
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)**
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

**Examples (different from previous)…**
Weak Coupling Example 1

Design-time coupling

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

- Simulator module calls few functions in Airplane
- Weak design-time coupling
Weak Coupling Example 2

Run-time coupling

- Client module makes many calls to Collection module
- Strong run-time coupling

- Client module makes few calls to Collection module
- Weak run-time coupling
Weak Coupling Example 3

Maintenance-time coupling

- 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 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)
A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion
- Has weak coupling