Modules, Interfaces and Implementations

- A big program is made up of many small ***modules***
- Each module implements (does) ***one*** thing
  - Mathematical functions
  - A hash table
  - A stack
- ***Interfaces*** specify ***what*** a module does
- ***Implementations*** specify ***how*** a module does it

More on Interfaces and Implementations

- ***One*** interface, perhaps ***many*** implementations
- Interface and its implementations must ***agree***
- ***Clients*** need see ***only*** the interface
  - may have only the object code for an implementation
- Clients ***share*** an interface and its implementations
  - avoids duplication and bugs --- implemented ***once***, used ***often***

Relationship

Clients

foo.c

```
#include "stack.h"

stack_push(s, x);
```

bar.c

```
#include "stack.h"

stack_push(s, x);
```

Interface

stack.h

```
typedef struct Stack_Rep_T *Stack_T;
extern void stack_push (Stack_T stk, void *x);
...
```

Implementations

stack.c

```
#include "stack.h"

void stack_push
    (Stack_T stk, void *x)
{ ...
};
```

my_stack.c

```
#include "stack.h"
...
```


Interfaces

- Modules ***export*** interfaces, clients ***import*** them
- Interfaces specify what clients may use or read
  - Data types, variables, functions, text specifications, ...
- They ***hide*** implementation details and algorithms
- In C, an interface is a ***single*** "`.h`" file; e.g. **`#include stack.h`**
- Interfaces are ***contracts*** between their implementations and clients

Client responsibilities	:	rules clients must follow to ensure correctness
Checked runtime errors	:	implementations guarantee to detect them
Unchecked runtime errors	:	implementations might not detect them
Performance criteria	:	implementations must meet them

- Checked/unchecked runtime errors are ***bugs***, not "normal user errors"

Implementations

- Implementations instantiate an interface
- In C, implementations are in "**.c**" files
- The ***interface*** is the key
- Some good things to do:
  - ***Decouple*** **clients** from implementations
    - Changes in an implementation do ***not*** affect clients
    - Implementations can be ***shared***, e.g. via libraries
  - ***Hide*** implementation details
    - Prevents dependency on specific representations and algorithms
  - ***Separate*** use of an interface from its implementations
    - Read specifications, not "unreadable" programs

Abstract Data Types (ADTs)

- ***Abstract data type***
  - A data type,
  - plus operations on values of that type
- ***Data type*: a *class of values***

  integers, reals, binary search trees, lists of integers, lookup tables ...
- ***Abstract*:** Independent of internal representation
- Advantages
  - ***Restricts*** manipulation of the values to a set of specified operations
  - ***Hides*** how the ADT is represented
- A key idea behind object-oriented programming

An ADT Example

- The interface **stack.h** defines a stack ADT and its operations

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED
typedef struct Stack_Rep_T *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, void *x);
extern void *Stack_pop(Stack_T stk);
extern void Stack_free(Stack_T *stk);
/* It is a checked runtime error to pass a NULL Stack_T or Stack_T* to
any routine in this interface or call Stack_pop with an empty stack. */
#endif
```

- The type "**Stack_T**" is an ***opaque pointer*** type
  - Clients can pass a **Stack_T** around, but can't look inside one
- "**Stack_**" is a disambiguating prefix
  - A ***convention*** that helps avoid name collisions in large programs
- Question: What does "**#ifndef STACK_INCLUDED**" do?

An Implementation

- **stack.c**

```
#include <assert.h>
#include <stdlib.h>
#include "stack.h"

struct Stack_Rep_T { void *val; Stack_T next; };

Stack_T Stack_new(void) {
    Stack_T stk = calloc(1, sizeof *stk);
    assert(stk);
    return stk; }

int Stack_empty(Stack_T stk) {
    assert(stk);
    return stk->next == NULL; }

void Stack_push(Stack_T stk, void *x) {
    Stack_T t = malloc(sizeof *t);
    assert(t);
    assert(stk);
    t->val = x;
    t->next = stk->next;
    stk->next = t; }
```

An Implementation Cont'd

```
void *Stack_pop(Stack_T stk) {
    void *x;
    Stack_T s;

    assert(stk && stk->next);
    x = stk->next->val;
    s = stk->next;
    stk->next = stk->next->next;
    free(s);
    return x;
}


void Stack_free(Stack_T *stk) {
    Stack_T s;
    assert(stk && *stk);
    for ( ; *stk; *stk = s) {
        s = (*stk)->next;
        free(*stk);
    }
}
```


Sample Client

- **`test.c`** includes **`stack.h`**

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
int main(int argc, char *argv[]) {
    int i;
    Stack_T s = Stack_new();
    for (i = 1; i < argc; i++)
        Stack_push(s, argv[i]);
    while (!Stack_empty(s))
        printf("%s\n", Stack_pop(s));
    Stack_free(&s);
    return EXIT_SUCCESS;
}
```

- **`test.o`** is a client of **`stack.h`**
  change **`stack.h`** → must recompile **`test.c`**
- **`test.o`** is loaded with **`stack.o`**
  `lcc test.o stack.o`
- **`stack.o`** is a client of **`stack.h`**
  changing **`stack.h`** → must recompile **`stack.c`**

Assertions

- Checked runtime errors are ***bugs***, not "normal user errors"
- **`assert(`*e*`)`** issues a message and aborts the program if *e* is 0

```
int Stack_empty(T stk){
    assert(stk);
    return stk->next == NULL;
}
```

- **`assert.h`** (approximately):

```
#ifdef NDEBUG
#define assert(e) ((void)0)
#else
#define assert(e) ((void)((e)|| (fprintf(stderr, \
        "assertion failed: file %s, line %d\n", \
        __FILE__, __LINE__), abort(), 0)))
#endif
lcc -DNDEBUG foo.c ...
```

- **Be careful**
  Turns assertions ***off*** — generally a bad idea
  e may not be executed — don't put code with ***side effects*** in *e*

The Standard C Library

- The ANSI C interfaces (See K&R, Appendix B)

`assert.h`	assertions
`ctype.h`	character mappings
`errno.h`	error numbers
`float.h`	metrics for floating types
`limits.h`	metrics for integral types
`locale.h`	locale specifics
`math.h`	math functions
`setjmp.h`	non-local jumps
`signal.h`	signal handling
`stdarg.h`	variable length argument lists
`stddef.h`	standard definitions
`stdio.h`	standard I/O
`stdlib.h`	standard library functions
`string.h`	string functions
`time.h`	date/time functions

- An ANSI C ***library*** provides the implementations
- ***re-use***, don't ***re-implement***; use libraries

The Standard C Library, cont'd

- Utility functions **`stdlib.h`**:

```
atof, atoi, strtod, rand, qsort, getenv,
calloc, malloc, realloc, free, abort, exit, ...
```

- String handling **`string.h`**:

```
strcmp, strncmp, strcpy, strncpy
strcat, strncat, strchr, strrchr, strlen, ...
memcpy, memmove, memcmp, memset, memchr
```

- Character classification **`ctype.h`**:

```
isdigit, isalpha, isspace, ispunct,
isupper, islower, toupper, tolower, ...
```

- Mathematical functions **`math.h`**:

```
sin, cos, tan, asin, acos, atan, atan2, ceil, floor, fabs
sinh, cosh, tanh, exp, log, log10, pow, sqrt,
```

- Variable-length argument lists **`stdarg.h`**:

```
va_list, va_start, va_arg, va_end
```

- Non-local jumps **`setjmp.h`**:

```
jmp_buf, setjmp, longjmp
```


The Standard I/O Library

- **`stdio.h`** specifies a **`FILE*`**, a good example of an ADT

```
extern FILE *stdin, *stdout, *stderr;
extern int fclose(FILE *);
extern FILE *fopen(const char *, const char *);
extern int fprintf(FILE *, const char *, ...);
extern int fscanf(FILE *, const char *, ...);
extern int printf(const char *, ...);
extern int scanf(const char *, ...);
extern int sprintf(char *, const char *, ...);
extern int sscanf(const char *, const char *, ...);
extern int fgetc(FILE *);
extern char *fgets(char *, int, FILE *);
extern int fputc(int, FILE *);
extern int fputs(const char *, FILE *);
extern int getc(FILE *);
extern int getchar(void);
extern char *gets(char *);
extern int putc(int, FILE *);
extern int putchar(int);
extern int puts(const char *);
extern int ungetc(int, FILE *);
extern int feof(FILE *);
```

- **Do you need to know what a `FILE* looks like`?**