Modules, Interfaces, and Abstractions

CS 217

Modularity

- Big programs are made up of many modules
- Each module does one thing…
  - mathematical function, symbol table, stack,…
- A module is to a large program what a procedure is to a CS126 assignment
  - a module may be implemented by many procedures
- Key difference: each module is designed to support potentially many users

Interfaces

- An interface defines what the module does
- An implementation defines how the module does it
- Each module has one interface, but potentially many implementations
  - efficiency (different algorithm)
  - machine dependencies
- Modules export interfaces, clients import them

Interfaces (cont)

- An interface is a contract between clients and the implementation
  - decouple clients from implementation
  - hide implementation details
- An interface specifies…
  - data types and variables
  - functions that may be invoked
  - client responsibilities
    - checked runtime errors
    - unchecked runtime errors

Interfaces in C

- Client

`user.c`

```
#include "stack.h"
main() {
   stack_push(x,  y);
}
```

- Interface

`stack.h`

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

- Implementation

`stack.c`

```
#include "stack.h"
void stack_push (
   Stack_T stk,
   void *item)
{ . . . }
```


Abstract Data Type (ADT)

- ADT: a kind of interface
  - a data type, plus…
  - operations on values of that type
- Data type: a class of values
  - integers, reals, search trees, lookup tables, sets,…
- Abstract because the class of values is independent of the internal representation
- Foundation of object-oriented programming
- Example: A Stack

stack.h

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

typedef struct Stack_T *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, void *item);
extern void *Stack_pop(Stack_T stk);
extern void Stack_empty(Stack_T *stk);

/* It's a checked runtime error to pass a NULL Stack_T
to any routine, or call Stack_pop with an empty stack */

#endif
```


Notes on **stack.h**

- Type **Stack_T** is an opaque pointer
  - clients can pass **Stack_T** around but can't look inside
- **Stack_** is a disambiguating prefix
  - a convention that helps avoid name collisions
- What does **#ifdef STACK_INCLUDE** do?

stack.c

```
#include <assert.h>
#include <stdlib.h>
#include "stack.h"
#define T Stack_T

struct T {void *val; T next };
T Stack_new(void) {T stk = calloc(1,sizeof *stk);
   assert(stk); return stk; }
int Stack_empty(T stk) {
   assert(stk); return stk->next == NULL; }
void Stack_push(T stk, void *item) {
   T t = malloc(sizeof *T); assert(t); assert(stk);
   t->val = item; t->next = stk->next; stk->next = t; }
void * Stack_pop(T stk) { void *x; 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(T * stk) { T s; assert(stk && *stk);
   for ( ; *stk; *stk = s) {
      s = (*stk)->next; free(*stk); }  }
```

user.c

```
#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;
}
```


Notes on `stack.c` & `user.c`

- Convention: **T** is an abbreviation for ***X*_T** for ADT ***X***
- **`user.o`** is a client of **`stack.h`**
  - change **`stack.h`** → must re-compile **`user.c`**
- **`user.o`** is loaded with **`stack.o`**
  - **`gcc user.o stack.o`**
- **`stack.o`** is a client of **`stack.h`**
  - change **`stack.h`** → must re-compile **`stack.c`**

Assertions

- **`assert(`*e*`)`**
  - issues a message and aborts the program if ***e*** is 0
  - need to include **`assert.h`**
- Activated conditionally
  - **`gcc –DNDEBUG foo.c`**
  - don't put code with side-effects in assertions
- Don't want to crash without a diagnostic
  - as you debug your code (assert invariants)
  - as someone else debugs their code that uses your code
  - as your code runs in production mode on random input

Standard C Libraries

assert.h	assertions
ctype.h	character mappings
errno.h	error numbers
math.h	math functions
limits.h	metrics for ints
signal.h	signal handling
stdarg.h	variable length arg lists
stddef.h	standard definitions
stdio.h	standard I/O
stdlib.h	standard library functions
string.h	string functions
time.h	date/type functions

Libraries (cont)

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

  `atof, atoi, rand, qsort, getenv, calloc, malloc, free, abort, exit`
- String handling **`string.h`**

  `strcmp, strncmp, strcpy, strncpy, strcat, strncat, strchr, strlen, memcpy, memcmp`
- Character classifications **`ctypes.h`**

  `isdigit, isalpha, isspace, isupper, islower`
- Mathematical functions **`math.h`**

  `sin, cos, tan, ceil, floor, exp, log, sqrt`

Standard I/O Library

- **`stdio.h`** defines **`FILE*,`** an example ADT

```
extern FILE *stdin, *stdout, *stderr;
extern FILE *fopen(const char *, const char *);
extern int fclose(FILE *);
extern int printf(const char *, …);
extern int scanf(const char *, …);
extern int fgetc(FILE *);
extern char *fgets(char *, int, FILE *);
extern int getc(FILE *);
extern int getchar(void);
extern char *gets(char *);
. . .
extern int feof(FILE *);
```

Archive Facility

- Creating a library

```
gcc –c stack.c –o stack.o
ar –rs mylib.a stack.o
```

- Using a library
  - include the interface specification (**`stack.h`**)
  - link against the archive (**`gcc user.c mylib.a`**)
- To see archive index

```
nm –s mylib.a
```