About COS 217

- Goals:
  - Prepare for other CS courses (and summer jobs)
  - Learn everything you need to know about ANSI C
  - **Master the art of programming**
    - design method, abstraction, interfaces and implementations, style
    - writing efficient programs
- Introduction to aspects of other courses
  - Low-level workings of a computer (more in COS 471))
    - SUN's SPARC architecture and instruction set
  - Assembly language programming (more in COS 320 and COS 471)
  - Operating systems (more in COS 318 and COS 461)
    - Programming using operating system services
  - Object-oriented programming

Everything is on the Web

- http://www.cs.princeton.edu/courses/cs217
  - Texts, Contact Information, Assignments, Lecture slides ...
- No handouts in class (except blank paper for quizzes)
- 9 assignments, including a final project
  - due on Monday at midnight. ***NO EXTENSIONS***.
- A few easy quizzes (15 min each, in-class)
- Midterm
- No final

This Course is About ...

- **Modules, interfaces and implementations**

```
Add_Box_To_Picture (Box,Picture,Position)
{
    ...
    ...
    Algorithm to implement function
    ...
    ...
}
```

```
Drawing_Program()
{
   ...
   do other things
   Add_Box_to_Picture(B,P,Pos)
   ...
   do other things
}
```

- What’s the module, interface, implementation, client?

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

Interfaces and Implementations: An Example

Driving an automobile

- Interface:
  - steering wheel
  - gears
  - brake
  - accelerator
  - clutch?
- Implementation:
  - engine and all its details

More on Interfaces and Implementations

- ***One*** interface, perhaps ***many*** implementations. Why?

    efficiency, different algorithms for different situations, machine dependences

- Interface and its implementations must ***agree***

- ***Clients*** need see ***only*** the interface

    do not need to understand implementation to use the module

    may have only the object code for an implementation

    • why might a client want to know more than the interface?

- Clients ***share*** interface and implementations

    avoids duplication and bugs --- implemented ***once***, used ***often***

- What does this sound like in your programming experience?

Client, Interface and Implementation: A Stack

Clients

user1.c

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

user2.c

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

Interface

stack.h

```
typedef struct Stack_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)
{ ...
```

mystack.c

```
#include "stack.h"

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

Interfaces

- Modules ***export*** interfaces, clients ***import*** them
- Interfaces specify what clients may use or read

  Data types, variables, function interfaces, text specifications, ...

  Everything a client needs to see
- They ***hide*** implementation details and algorithms
- In C, an interface is usually a ***single*** "`.h`" file; e.g. **`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, but they are bugs
Unchecked runtime errors	:	implementations might not detect them
Performance criteria	:	implementations must meet them

- Examples from the real world?

Implementations

- Implementations instantiate an interface
- In C, implementation source code is in “**`.c`**” files
- The ***interface*** is the key
- Some important things to do:
  - ***De-couple*** **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
    - User should read specifications, not programs

Abstract Data Types (ADTs)

- ***Abstract data type: A kind of interface***
  - A data type, plus
  - Operations on entities ("variables") of that type
- ***Data type*: a *class of values***

  integers, reals, lists of integers, binary search trees, lookup tables ...
- ***Abstract*: Operations permitted are indept. 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
  - BUT GOOD PROGRAMMING PRACTICE REGARDLESS OF LANGUAGE

An ADT Example: A Stack Again

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

```
#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 *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 of the Stack ADT

- **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 *x) {
    T t = malloc(sizeof *t); assert(t); assert(stk);
    t->val = x; 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);
    }
```

- Convention: In implementation, “**T**” is abbreviation of “*X*_**T**” for ADT *X*.

A Sample Client of the Stack ADT

- **test.c** includes **stack.h** (so it can use the stack ADT)

```
#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**

  changing **stack.h** → must re-compile **test.c**

- **test.o** is loaded with **stack.o**

  lcc test.o stack.o

- **stack.o** is also a client of **stack.h**

  changing **stack.h** → must re-compile **stack.c**

Assertions

- Even checked runtime errors are ***bugs***
- **`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 using assertions**

  e may not be executed if assertions are turned ***off*** (why would you do this?)

  — don't put code with ***side effects*** in an assertion

- Don't want program to crash without a diagnostic (safe programming)

Programming Style

- Variable names, indentation, program structure... Why?
- Who reads your programs?
  - compiler
  - users
  - other programmers
- Which ones care about style?
- Which ones do you program for?
- Difference between "macho" programmer and good programmer
- We'll talk more about style later

The Standard C Library Interfaces

- The ANSI C interfaces (See H&S, Ch 10)

`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

Libraries

- So why don't people always just use libraries?

- It's a great idea, but often not implemented well
  - Efficiency
  - Specific functionality
  - Mastering big libraries is hard
  - Library design is difficult: generality, simplicity and efficiency
  - Libraries may have implementation bugs

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`?**