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 systems courses
    • Computer Architecture (more in COS 471)
      - SPARC architecture and instruction set
    • Compilers (more in COS 320)
      - Assembly language programming
    • Operating Systems (more in COS 318 and 461)
      - Programming using operating system services

Everything is on the Web

• http://www.cs.princeton.edu/courses/cs217
  • Texts, Contact Information, Assignments, Lecture slides ...

• Many people have contributed over the years
  • Dave Hanson, Kai Li, JP Singh, Ann Rogers, Perry Cook

• No handouts in class (except blank paper for quizzes)

• 8 or so assignments (last one is a major project)

• Bi-weekly, easy quizzes (15-20 min each)

• Midterm

• Probably no final, no guarantees
This Course is About ...

- Modules, interfaces and implementations

```c
Add_Box_To_Picture (Box, Picture, Position) { 
    ... 
    ... 
    Algorithm to implement function Add_Box_to_Picture(B,P,Pos) 
    ... 
    ... 
} 
```

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

## Interfaces and Implementations

- A big program is made up of many smaller modules

- Each module implements (does) **one** thing
  
  - mathematical functions
  - hash table
  - stack

- An **Interface** specifies **what** a module does

- An **Implementation** specifies **how** a module does it
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

- 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**

```c
#include "stack.h"

main(){
    stack_push(s, x);
}
```

```c
#include "stack.h"

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

```c
#include "stack.h"

main(){
    stack_push(a, y);
}
```

---

```c
typedef struct Stack_T *Stack_T;

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

---

```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 a **single** “.h” file; e.g. `stack.h`

- Interfaces are **contracts** between their implementations and clients
  
<table>
<thead>
<tr>
<th>Client responsibilities</th>
<th>rules clients must follow to ensure correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checked runtime errors</td>
<td>implementations guarantee to detect them, but they are bugs</td>
</tr>
<tr>
<td>Unchecked runtime errors</td>
<td>implementations might not detect them</td>
</tr>
<tr>
<td>Performance criteria</td>
<td>implementations must meet them</td>
</tr>
</tbody>
</table>

- Examples from the real world?

Implementations

- Implementations instantiate an interface

- In C, implementations are 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 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: A Stack

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

  ```c
  #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
  
  #if defined 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);
  #endif
  
  #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
  
  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. */
  
  #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: “T” is an abbreviation of “X_T” for ADT X.

A Sample Client of the Stack ADT

• 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 re-compile test.c

• test.o is loaded with stack.o
  lcc test.o stack.o

• stack.o is a client of stack.h
  change 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

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

• `assert.h` (approximately):

```c
#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
```

• Be careful using assertions
  `e` may not be executed if assertions are turned **off** (why would you do it?)
  — 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