The exam was a closed-book, closed-notes, 50 minute exam.

**Question 1 Part a**

False. A "break" statement transfers control over the nearest surrounding "while", "for", or "switch" statement, but not over the nearest surrounding "if" statement.

**Question 1 Part b**

The first typedef defines "Node1" as an alias for "struct listnode *" and "Node2" as an alias for "struct listnode *".

The second typedef defines "Node1" as an alias for "struct listnode *" and "Node2" as an alias for "struct listnode".

**Question 1 Part c**

The given code is not good. It assigns 1 to some arbitrary spot in memory.

**Question 1 Part d**

(i) int*

Explanation: (a+1) is the same as (&a[0]+1). &a[0] is of type int*. So (&a[0]+1) also is of type int*.

(ii) *(a+1)

**Question 1 Part e**

A .o file should not depend upon another .o file.
A .o file should depend upon exactly one .c file, not upon multiple .c files.

**Question 1 Part f**

False. If test_all.c #includes test_all.h, then the makefile must express the fact that test_all.o depends upon both test_all.c and test_all.h.

**Question 1 Part g**

As given on slide 27 of the "Modularity" lecture...

**Question 1**

Is the function necessary to make the module complete?

**Question 2**

Is the function convenient for many clients?

**Question 2 Part a**

(i) 3696
Explanation: On our system, sizeof(float) is 4. So the statement "p = p + 4" adds 4*sizeof(float) = 16 to p, and assigns the result to p.

(ii) p = (float*)((char*)p + 4);

**Question 2 Part b**

int f(int *a);

**Question 2 Part c**

(1) The compiler generates a warning because it encounters a call of printf() before it sees a declaration or definition of printf().

(2) The compiler generates a warning because the function doesn't return a value of type int.

**Question 2 Part d**

1. A
2. A
3. A
4. B: The statement is OK at compile-time and run-time. However, since p doesn't point to the 0th element of an array, the statement is better expressed as "*p = 1;" We also accepted A as a correct answer.
5. D: The statement assigns 20 to some arbitrary spot in memory. The result could be a run-time error.
6. C: The name of an array is a constant pointer.
7. A
8. D: Failure to assign the value return by realloc() to a1 could corrupt a1, and thereby could cause a run-time error. We also accepted B, just because it happens that a1 never is dereferenced in the remainder of the function.
9. C: The name of an array is a constant pointer.
10. A
11. A

**Question 2 Part e**

The first directive asks "is XYZ defined." The second directive asks "is XYZ defined to some non-zero value."

#define XYZ 0

**Question 3 Part a**

(i)

int Stack_areEqual(Stack_T s1, Stack_T s2,
                   int (*f)(const void *item1, const void *item2));

(ii)

The strcmp() function is declared like this:

    int strcmp(const char *str1, const char *str2);
That declaration does not match exactly the type of the third formal parameter of Stack_areEqual().

(iii)

To avoid a compile-time warning, the client would need to cast strcmp such that its type matches that of the third formal parameter of Stack_areEqual(). So this would be an appropriate call:

```c
if (Stack_areEqual(s1, s2, (int(*)(const void*, const void*))strcmp)) {...}
```

**Question 3 Part b**

(i)

```c
int main(void)
{
    int a = 5; int b = 1;
    IncrementAndSwap(&a, &b);
    printf("a = %d\nb = %d\n", a, b);
    return 0;
}
```

```c
void IncrementAndSwap(int *x, int *y)
{
    (*x)++;
    (*y)++;
    Swap(x, y);
}
```

(ii)

At point 1 the value of x is the address of a. The value of y is the address of b. At point 2 x and y, the formal parameters of IncrementAndSwap(), no longer exist. So we cannot speak of their values.

**Question 4 Part a**

node_1 is a pointer to a constant. So it is possible for the function to change the value of node_1, but it is impossible (without casting away constness) for the function to change the value of the structure to which node_1 points.

**Question 4 Part b**

If the program is built with the NDEBUG macro defined, then calls of the assert() will be disabled. In that case the call of f() never will occur. That would affect the behavior of the program if the execution of f() causes side effects.

```c
int i;
...
i = f();
assert(i == 0);
```

**Question 4 Part c**

Functions that are not declared in a module’s interface should be defined as
"static" so clients cannot call them.

**Question 4 Part d**

This is the DFA expressed in textual form:

- **start state**: (the start state)
  - `a`: `a1_state`
- **a1_state**:
  - `a`: `a2_state`
  - `b,c`: `bc_state`
  - `d`: `accept_state`
- **a2_state**:
  - `a`: `a2_state`
  - `b,c`: `bc_state`
  - `d`: `accept_state`
- **a3_state**:
  - `b,c`: `bc_state`
  - `d`: `accept_state`
- **bc_state**:
  - `b,c`: `bc_state`
  - `d`: `accept_state`
- **accept state**:
  - `d`: `accept_state`

If no transition is possible, then the given string is not accepted. An alternative would be to define a failure state, and then define transitions labeled "other" from every state to the failure state.

**Question 5 Part a**

```c
void strcpy1(char dest[], const char source[]) {
    int i = 0;
    for (;;) {
        dest[i] = source[i];
        if (source[i] == '\0') break;
        i++;
    }
}
```

**Question 5 Part b**

```c
void strcpy2(char dest[], const char source[]) {
    int i = 0;
    while ((dest[i] = source[i]) != '\0')
        i++;
}
```

or

```c
void strcpy2(char dest[], const char source[]) {
    int i = 0;
    while (dest[i] = source[i])
        i++;
}
```
Question 5 Part c

void strcpy3(char dest[], const char source[]) {
    while ((*dest++ = *source++) != '\0');
}

or

void strcpy3(char dest[], const char source[]) {
    while (*dest++ = *source++);
}

Question 5 Part d

(i) By way of example... If the original program consumes 40 seconds, then 20 seconds were spent in the loop. In the optimized program, the loop consumes 5 seconds. So the optimized program consumes 20 + 5 = 25 seconds. So the execution time of the optimized program is 25/40, or 5/8 of the execution time of the original program. So the reduction in execution time is 1 - 5/8 = 3/8 of the original execution time.

(ii) A substantial change in the input data set might cause the original program to spend less than half of its time in the loop. As a result, the optimization might be less effective.

Copyright © 2012 by Jaswinder Pal Singh, Junjun Chen, Matthew Colen, Margo Flynn, Madhuvanthi Jayakumar, Sasha Koruga, Siyu Liu, Akshay Mittal, Tobechukwu Nwanna, Reid Oda, and Robert M. Dondero, Jr.