CS240 Final Solution 08/01/2018

P1(a) 12 pts

```c
y = (struct person *) malloc(sizeof(struct person));
```

4 pts

```c
x.name = (char *) malloc(sizeof("Smith"));
y->name = (char *) malloc(sizeof("Smith"));
```

4 pts

```c
x.age = 22;
y->age = 22;
```

2 pts

```c
strcpy(x.name, "Smith");
strcpy(y->name, "Smith");
```

2 pts

[Any valid assignment, including brute-force, to x.name and y->name are fine.]

P1(b) 12 pts

const char * prompts the C compiler (or gcc) to check that the code of myfunc() does not modify the characters pointed to by the argument of the function. The value of the argument, an address, is allowed to change. This provides some assurance to the programmer who uses myfunc() that the content pointed to by the argument will not be changed/corrupted by myfunc().

4 pts

```c
char *const means that the argument, a pointer containing an address, cannot be changed but the content pointed to by the argument can.
```

4 pts

Replacing struct with union has the effect of allocating memory for the largest field, in this case double which is 8 bytes. In these 8 bytes, either x of type float or y of type double can be stored, but not both.

2 pts

```c
In case one of the fields is used at a time, union allows the same memory to be used to store variables of different types. This saves memory compared to using struct (8 vs. 12 bytes in the example).
```

2 pts

P1(c) 12 pts

The first call returns 6, the second call returns 6.

4 pts

This is so since e is a local variable that resides in the stack frame of the function colts() which gets allocated when colts() is called and deallocated when it returns.

4 pts

When int e = 5 is made global, the first call returns 6, the second call 7. This is so since global variables reside above the code/text segment in memory which remains valid until a process terminates.

4 pts

P1(d) 12 pts

```c
// 3 pts
main(int argc, char **argv) {
FILE *fp;
int count = 0;
```
fp = fopen(argv[1],"r");
if(fp == NULL) {
    fprintf(stderr,"file cannot be opened\n");
    exit(1);
}

// 4 pts
while(fgetc(fp) != EOF)
    count++;

// 2 pts
printf("%d\n", count);
}

P2(a) 16 pts

main() {
    int x, y, i, j;
    char **mytab;

    // 2 pts
    scanf("%d %d", &x, &y);

    // 6 pts
    mytab = (char **) malloc(x * sizeof(char *));

    // 6 pts
    for(i=0; i<x; i++)
        *(mytab + i) = (char *) malloc(y * sizeof(char));

    // 2 pts
    for(i=0; i<x; i++)
        for(j=0; j<y; j++)
            scanf("%c", &mytab[i][j]);
}

P2(b) 16 pts

system() takes a string as argument that specifies the executable to be executed. Since system() returns to the process that made the function call, system() must,
2 pts
one, fork() to clone the process,
4 pts
and two, call exec() (i.e., one of the exec-family) in the child process to load,
replace and execute the cloned child process with the specified executable.
4 pts
The parent process returns from system() whereas the child process, if fork() and
exec() are successful, does not return from system().
4 pts
Any command-line arguments specified in the string input to system() are passed
onto exec() in the child process.
2 pts

P3 20 pts

int cprintf(char *a, ...) {

    // 2 pts
    int i, count = 0;
    char s;
    va_list arglist;
// 2 pts
va_start(arglist, a);

// 8 pts
while(*a != '\0') {
  if ((*a == '%') && *(a+1) == 'c')
    count++;
  a++;
}

// 6 pts
for (i=0; i<count; i++) {
  s = va_arg(arglist, char);
  write(1, &s, 1);
}

// 2 pts
va_end(arglist);

Bonus 10 pts

If fork() succeeds, it returns 0 in the child process and the PID of the child in the parent process. Hence, fork() can be viewed as returning twice.

5 pts

If exec() (i.e., one of the exec-family) succeeds, the calling process's image (code and data) is replaced by the executable specified in the argument of exec(). The executable commences at its first instruction. Hence exec(), if successful, does not return.

5 pts