CSE 333 – Autumn 2014
Midterm Key

1. *[3 points]*
Imagine we have the following function declaration:

```
void sub(uint64_t A, uint64_t B[], struct c_st C);
```

An experienced C programmer writes a correct invocation:

```
sub(a, b, c);
```

Briefly explain what values in the calling code the programmer might reasonably expect could be changed by that call. For instance, might the value of `a` change?

None of the variables a, b, or c can have their values changed, because C is purely call-by-value. On the other hand, the caller can examine things that b refers to (e.g., b[3]), and the callee can modify those things, so they may be changed on return.
(Additionally, c is a struct, and we don't know what it's fields are. If any of them are pointers, what they point to can be changed by the callee, and that change will be visible in the caller. There were bonus points for mentioning this, but it wasn't required to get full credit.)

2. *[4 points]*
This statement defines a variable, `Q2`:

```
struct {
      int   intField;
      char  charField;
      void  *pointerField;
} Q2;
```

(A) What is the value of each of the following?

`sizeof(Q2.intField)`	____4___________________
`sizeof(Q2.charField)`	____1___________________
`sizeof(Q2.pointerField)`	____8 (64-bit) or 4 (32-bit)_

(B) Briefly explain why `sizeof(Q2)` might not be equal to the sum of those three terms.

The compiler must lay out the fields in the ordered declared. It may choose to put a field at an address that is a multiple of that field's length, for performance reasons. (This is called alignment.) After laying out the int and the char, the compiler may insert "padding" so that the pointer field is at an address that is a multiple of the pointer's length (8). The struct therefore can include padding, so its total length can be larger than the sum of its fields' lengths.

3. *[8 points]*
Imagine we're implementing a variable sized array of integers as "class" `Array`. An `Array` has a capacity, which is how many elements it can currently hold, and a size, which is how many it is currently holding. Here's a definition of the `Array` type and the bit of its implementation that creates a new instance:

```
typedef struct {
      int capacity;
      int size;
      int *values;
} *Array;

Array array_new() {
 int values[10];
 Array a = (Array)malloc(1 * sizeof(Array));
 a->capacity = 10;
 a->size = 0;
 a->values = values;
 return a;
}
```

The `array_new` function above compiles fine but is buggy. Write a bug-free version here.

```
Array array_new() {
      Array a = (Array)malloc(1 * sizeof(*a));
      if ( a == NULL ) return NULL;

      a->values = (int*)malloc(10 * sizeof(int));
      if ( a->values == NULL ) {
            free(a);
            return NULL;
      }

      a->capacity = 10;
      a->size = 0;

      return a;
}
```

4. *[2 points]*
It's almost always an error to put code (for example, the definition of a subroutine) in a `.h` file because (pick one):

(a) unless you go way out of your way, the preprocessor won't recognize the code and will generate errors

(b) modern compilers simply won't allow code that comes from .h files

(c) unless you're lucky, the linker will generate errors

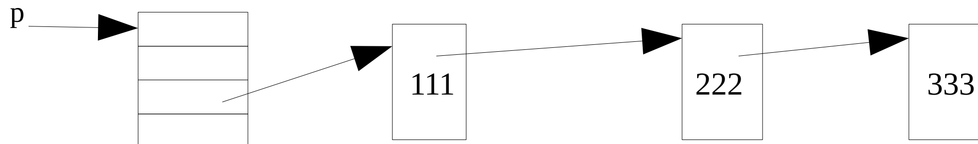
(d) the code will preprocess, compile, and link, but you'll get run time errors when you try to execute because the code in the .h will be in the static data segment, which isn't executable

The answer is (c). (a) isn't right because the preprocessor processes #xxxxx directives but simply passes all other text through. (b) isn't right because the compiler can't tell where the code came from. (c) is right because it's common for there to be multiple source files, and if each one #includes some code that code will have multiple definitions. (The compiler won't detect that because it compiles a single file at a time. The linker will notice when it tries to put all the .o files together to build the executable.) (d) isn't right because everything about it is wrong.

5. *[3 points]*
Suppose you have the following variable declaration:

```
typedef struct element_st {
      struct element_st *next;
      int val;
} Element;
Element *p[4];
```

And suppose that you know that the data currently looks like this:



Write a C statement that will change the value 222 to 0.

p[2]->next->val = 0;

p[2] is a pointer to an Element. Take that element's next field and dereference it (because of the ->). That's an Element. Now take that element's val field and set it to 0.

6. *[2 points]*
Suppose I've written method `sub()` in file sub.c, and I want to call it from code in file `main.c.` In that case I can either:
(a) put a prototype declaration for `sub()` near the head of `main.c`, or
(b) put a prototype delcaration for `sub()` in file `sub.h` and `#include sub.h` near the head of `main.c`

Briefly, but precisely, explain why (b) is a much better solution than (a).

We want the definition of sub (in the .c file) and the declarations of sub (required by the uses of sub) to either be consistent or for the compiler to detect that something is wrong. If each file using sub has its own declaration simply typed into that file, if the sub interface changes (e.g., a parameter is added), the source files using sub will still

compile fine even if their declarations of sub are not changed – the compiler will just believe the declaration typed into those files and generate code assuming they're right. If both the uses of sub and the definition of sub share a single declaration of sub by #include'ing the .h file, then if they disagree about sub's interface at least one of them will get compile time errors because the use (or definition) of sub won't match the declaration.

7. *[4 points]*
I'd like to be able to write statements like this:

```
#include "isvowel.h"
...
if ( isVowel(c) ) printf ("%c is a vowel", c);
```

where c is of type `char`. `isVowel` returns true if the character is a vowel (a, e, i, or u) and false otherwise. For performance reasons, I want `isVowel` to be implemented as a preprocessor macro.

Write a version of the `isvowel.h` file that satisfies my requirements.

This questions asks you to write the file (so #ifndef guard code is needed):

```
#ifndef ISVOWEL_H
#define ISVOWEL_H

#define isVowel(c) ((c)=='a' || (c)=='e' || (c)=='i' || (c)=='o'
|| (c)=='u')

#endif // ISVOWEL_H
```

Note that the preprocessor is going to substitute the string value #define'd for isVowel(c), substituting whatever the calling code has supplied for 'c', directly into the calling code. You cannot have a return statement, or any statements, in that string value if you want the example code to compile. (If you do have a return, you'll end up with "if (…. return...) ..." as what the compiler sees for the example code, which wont be good.)

8. *[8 points]*
Use an assumed correct version of `isvowel.h` to write a subroutine that counts the number of vowels and non-vowels in a file whose name is passed as the only argument. Your subroutine should return those two counts by returning a single struct of this type:

```
typedef struct isvowel_result_st {
      int isCount;
      int isntCount;
} IsVowel_result;
```

(If any errors occur, you should return counts of -1.)

```
IsVowel_result countVowels(const char *filename) {
      IsVowel_result result = { .isCount = 0, .isntCount = 0 };

      FILE *f = fopen(filename, "r");
      if (f == NULL) {
            result.isCount = result.isntCount = -1;
```

```
            return result;
        }

        char c;
        while ( (c=fgetc(f)) != EOF) {
              if ( isVowel(c) ) result.isCount++;
              else result.isntCount++;
        }
        fclose(f);

        return result;
    }
```

(IF you wanted to be really client-code friendly, you'd check that filename != NULL before trying to use it for anything, and return an error if it were. This code (likely) just causes a segfault in a way that's harder for the client to debug.)

9. *[6 points]*
   Now build on the code you wrote in the last two question to write an application that prints out the number of vowels and the number of non-vowels in the files whose names are given as command line arguments, e.g.,
   `$ ./countvowels countvowels.c isvowel.h somefile.txt`
   (Assume that the solution to question 8 provides a .h file with a suitable prototype declaration to make use of your subroutine implementation.)

```
int main(int argc, char *argv[]) {
   // Note that I don't have to check argc here...
   for (int i=1; i<argc; i++) {
         printf("\nFile: %s\n", argv[i]);
         IsVowel_result result = countVowels(argv[i]);
         if ( result.isCount < 0 )
               printf("Some error: probably couldn't open %s\n",
                      argv[i]);
         else
               printf("\tisCount / isntCount = %d %d\n",
                      result.isCount, result.isntCount);
   }
   return 0;
}
```

There's the issue of whether the spec is ambiguous – should the app report separate results for each file listed on the command line (as this solution does), or should it report a total for all the files? Modularity suggests the former. The user can always cat *multiple files into one if that's what they want. If the app effectively does the cat, it's awkward to use if that isn't what you want.*

Aside: Typical Unix apps will allow a file name of '-' to mean stdin. So, I could write something like this:
$ cat countvowels.c isvowel.h somefile.txt | ./countvowels -

if I want the total number of vowels in a set of files. (Sometimes the absence of any file name will mean read from stdin, rather than having to type the '-*'.* cat *itself does that, for instance.)*