CSE 333 – Autumn 2015
Midterm Key

1. *[3+4+2=9 points]*
`map()` is a function that takes a function as an argument and applies it to each element of some collection. The calling code hands `map()` the collection and a function. `map()` successively invokes the function on each element of the collection. `map()` returns nothing.

   In this question you're asked to implement a specific version of `map()` where the collection is an array of `void*` pointers. This being C, the collection is really just an array of any data type that can be stored in 8 bytes – only the user of the `map()` code knows what the actual type of the array elements might be.

   (A) Give a declaration for `map()`. Its arguments are the information needed for it to iterate over the `void*` array and a function to be invoked on each element in that array. The argument function takes a single argument and returns nothing. **It is invoked in a way that lets it change the value of that element in the array.** (Part (B) gives an example use, which might help clarify the specification.)

   Reminder of function type definition syntax:

   ```
   typedef int (*IntFunc)(int x);
   ```

   makes a new type, `IntFunc`, that is a function taking a single `int` argument and returning an `int`.

   ```
   typedef void (*mapFn)(void **element);
   void map(void *array[], int size, mapFn fn)
   ```

   (B) Now imagine applying `map()` to `argv` in `main()` so that afterwards `argv` contains the original number of non-NULL elements but each now points to the string "REPLACED". Write a function, `reset()`, that can be used as an argument to `map()` and will achieve the desired result.

   ```
   void reset(void **element) {
    *(char**)element = "REPLACED";
   }
   ```

   (C) Write the line of code that would appear in `main()` to invoke `reset()` on each element of `argv`.

   ```
   map((void**)argv, argc, reset);
   ```

2. *[6 points]*
I have tried to write a subroutine that takes a file name and a size and returns a string containing the initial size characters of the file. Here's my code:

```
#include <stdio.h>
#include <stdlib.h>
char *fileHead(char *filename, int nChars) {
   int i;
   char *head = (char*)malloc(sizeof(char) * nChars);
   FILE *f = fopen(filename, "r");
   if ( f == NULL ) return NULL;
```

```
   for ( i=0; i<nChars; i++ ) {
     head[i] = fgetc(f);
     if ( head[i] == EOF ) break;
   }
   fclose(f);
   return head;
}
```

My code compiles without warnings, and when I run simple tests it produces the expected results. However, my code has several serious bugs. Briefly describe two of them.

- Memory leak – malloc succeeds but file open has an error, memory is lost.
- Doesn't check return value from malloc
- Doesn't create a C string – there's no guarantee there will be a '\0' in the returned buffer

3. *[3 points]*

Add code between the two lines below that initializes `p` so that the result of the code snippet is that "Q3" is printed.

```
char *p;

strcpy((char*)&p, "Q3");

printf("%s\n", (char*)&p);
```

4. *[8 points]*

Provide the code for the subroutine below. (Your code would go where the subroutine appears in this file, but you'll write it below this question.) Your implementation should meet the "specifications" given by the results printed for the test commands shown after the code.

```
#include <stdio.h>
#include <stdlib.h>
long int hexStringToLong(char *x) {
 <your code would be here>
}
int main(int argc, char *argv[]) {
 if ( argc != 2 ) {
 fprintf(stderr, "Usage: %s <hex integer>\n", argv[0]);
 exit(1);
 }
 long int x = hexStringToLong(argv[1]);
 printf("%lx\n", x);
 return 0;
}
```

Example invocations:

```
$ ./q4 01A2b
1a2b
$ ./q4 123z
123
$ ./q4 fedcba9876543210fedcba9876543210
fedcba9876543210
```

Note: 'a' to 'f', 'A' to 'F', and '0' to '9' are each consecutive ranges of (integer) values.

```
long int result = 0;
while( *x ) {
      char base;
      if ( *x >= '0' && *x <= '9' )  base = '0';
      else if (*x >= 'A' && *x <= 'F' ) base = 'A'-10;
      else if ( *x >= 'a' && *x <= 'f' ) base = 'a'-10;
      else break;
      result = result * 16 + *x - base;
      x++;
}
return result;
```

5. *[3+2+1 = 6 points]*
Imagine that you are going to implement a generic binary tree data structure in C. The `tree` will consists of zero or more `nodes`.

(A) Define here a plausible type for the `nodes` of the `tree`.

```
typedef struct node_st {
   struct node_st *left;
   struct node_st *right;
   void           *payload;
} node;
```

(B) Define here a plausible type for a `tree`.

```
typedef node *tree;
```

(C) Based on your previous answers, give an expression that evaluates to true if the `tree` is empty (has no nodes).

```
(myTree == NULL)
```

6. *[4+2 = 6 points]*
(A) Design and implement a function whose purpose is to create a copy of an arbitrary array. So, for example, your function should be able to copy an array of integers or an array of doubles or an array of some struct type. (By "design," we mean the the parameters and return value of the function are up to you.)

```
void* copyArray(void* array, int size) {
   int i;
   char *newArray = (char*)malloc(size);
   if ( newArray == NULL ) return NULL;
   for (i=0; i<size; i++) {
         newArray[i] = ((char*)array)[i];
   }
   return newArray;
}
```

(B) Assuming MyStructType is some previously defined structure type, show how to invoke your function to create a copy of this array:

```
MyStructType array[N];

MyStructType *newArray =
    (MyStructType*) copyArray(array, N*sizeof(MyStructType));
```

7. *[3 points]*
Given this declaration:

```
char * sub(void *one, int two, int *three, char **four);
```

if I invoke sub as follows, which of the arguments I pass to it may have a different value on return than it had when I called sub?

```
p = sub(one, two, three, four);
```

None – C is call by value.