CS 451 / 551 Homework 1

Due: Feb 15 **Max Points:** 100

To be turned in on paper in class.

Please remember to justify all answers.

Important Reminder As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

In questions (2) and (3), assume that `Shape` has been `typedef`'d to a suitable `struct`.

1. On a particular C system a `short` occupies 2 bytes with an alignment restriction that it must be stored at an address which is divisible by 2. Discuss how you would proceed to tighten this alignment restriction to force a particular `short` entity to always be stored at an address divisible by 8. Document any assumptions in your solution. *5-points*

2. Describe in English the types of the following variables. Identify any types which are invalid: *15-points*

   a) `Shape* a, b;`

   b) `const Shape *(fs[])(double);`

   c) `const Shape (*fs[])(double);`

   d) `Shape *const p;`

   e) `int (*cmps[10])(const Shape *, const Shape *);`

3. A function `f()` takes a single argument and returns a single value. The type of the single argument to `f()` is a pointer to a function which takes an unspecified number of arguments and returns a pointer to a `const Shape`. The value returned by `f()` is a pointer to an array of pointers to functions which take a single `int` argument and return a single `Shape` result. *10-points*

   a) Give a declaration for `f()` using auxiliary `typedef`'s.

   b) Give a declaration for `f()` without using auxiliary `typedef`'s.

4. Identify bugs and inadequacies in the following function.

```
/** Return first line read from file named fileName.  The
 *  terminating newline is not returned.
 */
char *
getFirstLine(char *fileName)
{
   int maxLine = 1024;
   char line[maxLine];
   FILE *in = fopen(fileName, "r");
```

```
    char *p = line;
    fgets(line, maxLine, in);
    char *p = line;
    if (line[strlen(line)] != 10) { //line[] too small, alloc dynamically
      char *p = malloc(sizeof(2*maxLine));
      fgets(p, 2*maxLine, in);
    }
    p[strlen(line)] = 0;  //replace newline with NUL char
    return p;
}
```

You should assume that all required header files have been included. *15-points*

5. Given an instance of class `C` in a classical OOPL, instances of classes totally unrelated to `C` can have `public` access to the instance variables of `C`. The techniques discussed in class for simulating OOP in C in the oo-shapes example and prj1 do not allow such access, limiting access to the instance variables of C solely to the instances of C. How would you modify the techniques to allow access similar to `public`? *10-points*

6. How would you extend the techniques discussed in class for simulating OOP in C in the oo-shapes example and prj1 to allow for interfaces. Specifically, how would you make it possible for a client to take a `Shape` and treat is as a `DrawableInterface` where `DrawableInterface` will reference the following functions:

```
typedef struct {
  void (*draw)(DrawableInterface *drawable, GraphicsContext *ctx);
} DrawableFns;
```

   for some graphics context `ctx`.

   For example, there may be a inheritance hierarchy like `WireFrame` different from the `Shape` hierarchy which may also implement a `Drawable` interface.

   Like the rest of the techniques, using a `Shape` as a `DrawableInterface` may require that the client follow certain rules and need not be as seamless as in a real OOPL. *10-points*

7. Real OOPL's protect objects from invalid access. For example, given the Java `class`:

```
class Circle extends Shape {
  private final double radius;

  Circle(double r) { radius = r; }

  double getRadius() { return radius; }
}
```

   [For those unfamiliar with Java, the above declares `Circle` to be a subclass of `Shape` with all `Circle` instances having a single field `radius` which cannot be changed (`final`) once the instance has been constructed (using the `Circle()` constructor)].

With normal Java semantics, is impossible for a client of `Circle` to change the radius of a `Circle` instance once it has been constructed (ignoring reflection which makes a mockery of things like `final` declarations).

Since C is not a OOPL, it makes no such guarantees. Show how a client of `Circle` (implemented as in the `oo-shapes` example discussed in class) could modify the radius of a `Circle` instance after constructing it. Specifically, you should be able to show how you would achieve something like this:

```
const Circle *circle1 = newCircle(1);  //radius is 1
//your code here (cannot change circle1 pointer)
printf("created circle1 = circle(radius = %g)\n",
       circle1->fns->radius(circle1));
```

and have it print `created circle1 = circle(radius = 2).`

You may assume that the client has **read-only** access to all the source code implementing `Shape`'s, `Circle`'s, etc. *10-points*

8. Some languages permit *lazy evaluation*. How would you implement a restricted form of lazy evaluation in C. A motivating example follows:

   Consider an array of function pointers used as shown below:

```
int (*fns[])(int exp) = { ... };

int p = ...;
int s = 0;
for (int i = 0; i < sizeof(fns)/sizeof(fns[0]); i++) {
  int (*f)(int) = fns[i];
  int exp = ...; //compute exp(p) for some param p.
  s += f(exp);
}
```

   Unfortunately, under some situations, the computation for `exp` blows up (maybe a division by zero or referencing a `NULL` pointer). It turns out that the knowledge needed to implement a check to see if `exp(p)` is dangerous is not available at the level of the above code but is available within each individual function. But since `exp` is evaluated before the call to the function, using the check within the function would be too late. This could be avoided by using lazy evaluation of `exp`.

   How would you set things up so as to delay the evaluation of `exp(p)` until after an individual function has been able to run the check to ensure that it is safe. You may change both the above code as well as the interface and implementation of the individual functions in the array. *10-points*

9. Discuss the validity of the following statements: *15-points*

   a) If a C function is declared with prototype `void f()` but then defined with prototype `void f(int a)`, then the compiler will signal an error about the inconsistency in the prototypes.

   b) The `stderr` stream need not always be unbuffered.

c) It is fine to `malloc()` a single character at a time.

d) `NULL` and `0` are equivalent in pointer contexts.

e) In the standard I/O library on Unix systems, there is no noticeable difference between opening a file as binary versus opening it as text.