Aniruddha Tekade (B00618834) February 15, 2017

CS 551
Homework 1

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.

Assumption: The target system architecture has natural alignment support. we refer to as natural alignment: *When accessing N bytes of memory, the base memory address must be evenly divisible by N*, i.e. `addr % N == 0`.

Because, a system may have unaligned memory access i.e. we try to read N bytes of data starting from an address that is not evenly divisible by N (i.e. `addr % N != 0`). Therefore there are 2 ways by which we can achieve this :

- **u64** ⇒ with the above asumption, we can rely on compiler that it will always allocate memory to the short only at the addresses with a start divisible by 8. Because, `u8` is 1 byte and as per the requirement of this problem, we need an address start that is divisible by 8. So, wherever we are defining a `short`, we will always define it as `u64 short var;`.

  This is a safe way as compiler handles it and UNIX's Kernel Programming also includes this with guarantee the feature of proper alignment as per the requirement of the program code.

- **manual padding of bytes** ⇒ programmer has to define the number of padding bits so as to maintain the alignment of address divisible by 8. This is not an advisable way as the usual padding that is done is by allocating memory to `char` since it takes only 1 byte. But this technique may lead to memory leaks making programmer handel them first.

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

(a) `Shape* a, b;`
Explanation ⇒ `a` is pointer to the structure `Shape` and b is structure object. This is a `valid` statement provided structure definition is available. But `b` can become `invalid` if there is no stucture definition present and compiler will know that the the storage size of `b` is unknown. To resolve this issue we can either modify the statement as `Shape* a, * b;` or define the structure.

(b) `const Shape *(fs[])(double);`
Explanation ⇒ `Invalid Statement`. Because while attempting to decode the complex declaration, compiler will know that `fs` is an array to a function that takes single argument which of type dpuble. But the `*` specified before function `(fs[])` will generate an error as it expectes the return type of the function. Looking at the statement seems that it returns the stucture `Shape` but compiler throws an error because of illegal usage of pointer.

(c) `const Shape (*fs[])(double);`
Explanation ⇒ `fs` is array of pointer to a function that takes sing argument which is of type double and returns const structure `Shape`.

(d) `Shape *const p;`
Explanation ⇒ `p` is a const pointer to a structure `Shape` and is a legal legal declaration for compiler.

(e) `int (*cmps[10])(const Shape *, const Shape *);`
Explanation ⇒ `cmps` is an array of 10 pointers to a function which takes two arguments which are const pointer to the structure `Shape` and return an interger. This is a legal declaration.

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.

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

```
Shape *((*[])(int)) f(const Shape* (*) ())
```

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

```
struct *((*[])(int)) f(const struct* (*) ())
```

4: Identify bugs and inadequacies in the following function.

```
/** Return first line read from file named fileName.
* 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.
Following bugs and inadequacies are present in the above code sample -

- Pointer `p` is getting redefined which is not legal for compiler
- The pointer `p` that is being returned with a `0` at the end of the file which is not a good practice of file handling.
- Although it is not a problematic but having `0` instead of `NULL` at the end of the file can cause problems for future usage of the file.

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`?

Explanation ⇒ To modify the access to the instances of unrealted classes to `C`, I think we can take following simple possible approach -

1. Add a *function pointer* with unspecified number of arguments like `(*)()` as a member into the class `C`.
2. This will make the function pointer available to all the instances of class.
3. Define the classes and their respective instances and pass the *function pointer* to its methods. Automatically, the instances of the required class `C` will be accessible to the new instance of the class totally unrealted to C.

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 `DrawableInterface`.

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.

Explanation ⇒ To use `Shape` as an `DrawableInterface`, we should add the declaration specified in the question to the object of the inherited class. For instance if `Shape` is being inherited by the class `Circle`, then we add the pointer to function defining `DrawableInterface`. This is realized as follows -

- **Assumption** ⇒ `oo-shape` code discussed in the class is being considered as a basis to this question. Also, `Circle` is the derived class.
- We define a forward reference for the `DrawableInteface` in the `Shape.h` as follows -

```
struct DrawableFns; //forward reference

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

```
/*Adding reference of above structure into the Shape*/
typedef struct Shape {
    ShapeFns *fns;
    DrawableFns *dfns;
} Shape;
```

- Now, just in case, if `Circle.h` wants to use the `Shape` as its interface, it can create an object of its own and include reference to `Shape` as follows -

```
typedef struct CircleFns CircleFns;

typedef struct {
    CircleFns *fns;
} Circle;

struct CircleFns {
    ShapeFns;    //using -fms-extensions
    DrawableFns;     //using -fms-extension
};
```

- To access the above interface from anywhere, we can just create an object of the structure and use the `Shape` as `DrawableInterface`. For example -

```
typedef struct {
    Circle;    //using -fms-extension
    double radius;
} circleObject;
```

- We also define and access the method `draw()` in the `Circle.c` class -

```
void draw(DrawableInterface *drawable, GraphicsContext *ctx)
{
    // contains logic for drawing the shapes of different kind
}
```

7: Real OOPLs 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.

Explanation ⇒ This can be achieved by returning a new circle with specified radius by client as follows -

- First of all we will have to define a structure including `struct Circle` as follow -

  ```
  struct CircleObject {
      struct Circle; // this is allowed due to -fms-extension in C
      double radius;
  };
  ```

- And then defining the method that returns the radius of the constructed circle as follows -

  ```
  struct Circle* newCircle(double r){
      struct CircleObject *circle = malloc(sizeof(struct CircleObject));
      if(!cirlce) { //validation if malloc has failed
  ```

```
            // error message
        }
        circle->radius = r; //which is value requested by client of Circle
        return (struct Circle *) circle;
    }
```

- This ensures that any value requested by the client for the radius can be set up and the structre pointer is retured back.

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.

Explanation ⇒ The *lazy evaluation* can be forced by changing the type of argument of function `f()` to `void *` i.e a *generic pointer*. Becuase of this implementation, when the value of `exp(p)` is expcted, the evaluation will be delayed.

The altered code snippet will look like this -

```
int (*fns[])(void* exp) = { ... }; //changed type of argument

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

9: Discuss the validity of the following statements:

(a) Not Valid. Because when a function call is made to the function `f()`, the C compiler tries to match the definition of the function and the inconsistency is resolved. Redefining the prototype also widens the scope of the prototype in some conditions. Hence redefined prototype is acceptable.

(b) Valid. Because `stderr` returns just the pointer to the textual representation of the current error code/value. Secondly it is used with fprintf with the syntax `fprintf(stderr, <string>, <values to print> ...);`. So its the function `fprintf()`'s responsiblity to access the stream and print the error. Therefore there is not need to unbuffer the `stderr`.

(c) Not Valid. Because `malloc()` randomly allocates memory to the character variable. Although it is fine for a single character variable, this does not make helpful approach in the case an array of characters i.e. a `string`. So accessing the memory may become difficult and also it is inefficient.

(d) Valid. Because `NULL` is macro defined in `<stddef.h>` and several other standard header files that implements it as `null pointer constant`. So the expansion of `NULL` is either `0` or `((void *)0)`. Therefore a literal `0`, when used in a context of pointers, it always evaluates to a `NULL` pointer.

(e) Valid. Because both text and binary files are treated in the same way by Unix file handling mechanisms. However using the appropriate mode may improve the portability in the input and output operations for a particula type of file. Based on the environment in which a file gets opened, some special character conversion may occur in input/output operations in text mode of the file. This feature actually makes changes according to a system-specific file format. However some environment may not change it.