Pointers to functions

Pointer variables may also hold the address of a function and be used to invoke the function indirectly:

```
class/215/examples ==> gcc p32.c
class/215/examples ==> cat p32.c

/* p32.c */

int adder(
int a,
int b)
{
   return(a + b);
}

main()
{
   int (*ptrf)(int, int);  // declare pointer to function
   int sum;

   ptrf = adder;           // point it to adder (note no &)

   sum = (*ptrf)(3, 4);    // invoke it
   printf("sum = %d \n", sum);
}
class/215/examples ==> a.out
sum = 7
class/215/examples ==>
```

Function pointers as do-it-yourself polymorphism

Recall that the *sceneobj_t* structure contained the “hits” function pointer:

```
typedef struct sceneobj {
      /* Pointer to type dependent data */
         void    *priv;        /* Private type-dependent data            */

      …
      /* Hits function.            */
         double  (*hits)(double *base, double *dir, struct obj_type *);

      …
} sceneobj_t;
```

The *hits* pointers must be set in a scene object’s initialization module. For example, in the *sphere_init* function if new is pointing to a sphere’s associated obj_t structure the *hits* pointer should be set as follows:

```
      sceneobj_t *sceneobjPtr;
      …
      sceneobjPtr = new ->catpriv;
      …
      /* Set link to "hits" function */
      sceneobjPtr->hits = sphere_hits;
```

where the *sphere_hits()* function is declared as follows.

```
double  sphere_hits(
double  *base, /* the (x, y, z) coords of origin of the ray */
double  *dir,  /* the (x, y, z) direction of the ray        */
obj_t   *obj)  /* the object to be tested for the hit.      */
{


}
```

The *find_closest_object()* function.

When a ray is fired from the viewpoint through a pixel, depending upon the nature of the objects in the scene, the ray may pass through several objects or it may hit none at all. The color of the pixel will be derived from the *material* properties of the *first*object the ray hits.

The ray trace function should rely upon *find_closest_object* to return a point to the *closest* object that is hit by the ray. If *none* of the objects in the scene is hit, NULL must be returned. The distance from the base of the ray to the nearest hitpoint is returned via the *distance* field in the closest object's sceneobj_t structure.

```
   closest = find_closest_object(model->scene,
                                 base, dir, NULL);
```

The *find_closest_object()* function just processes the complete object list and calls the appropriate hits function for each object found.

```
   obj_t *obj;
   sceneobj_t *sceneobj;
   …
   obj = lst->head;
   while ((obj != NULL))
   {
      if (obj != last_hit)
      {
         sceneobj = obj->catpriv;
         dist = sceneobj->hits(base, dir, obj);
      :
      :
```

This approach provides a much cleaner and more efficient mechanism than the functionally equivalent way:

```
   obj = lst->head;
   while ((obj != NULL))
   {
       if (strcasecmp(obj->objclas, "sphere") == 0) {
           dist = sphere_hits(base, dir, obj);
       }
       else
       if (strcasecmp(obj->objclass, "plane") == 0) {
           dist = plane_hits(base, dir, obj);
           etc.
```

From the software reliability perspective an important advantage of this approach is that when a new object type is added *it is not necessary to modify find_closest_object().*

Tables of function pointers

The model dumping and loading operation provides an example in which tables of function pointers can replace a big messy switch construct and simplify the adding of new object types. The following example

1 - declares as a field within a structure array pointers to object constructor functions
2 - initializes the pointers to point to the appropriate functions

```
/**/
obj_t *error_init(FILE *in, char *objclass) {
   fprintf(stderr, "Unknown object class \"%s\"\n", objclass);
   return(NULL);
}

/**/
void error_dump(FILE *out, obj_t *objPtr) {
   fprintf(stderr, "Unknown object type \"%s\"\n", objPtr->objclass);
}

/* Supported objects */
struct {
    char *className;
    obj_t *(*classInit)();
    void (*classDump)();
} classes[] =
  {{"sphere", sphere_init, sphere_dump},
   {"plane", plane_init, plane_dump},
   {NULL, error_init, error_dump}};

/**/
int model_init(FILE    *in, model_t *model) {
   char  buf[256];
   char  objclass[128];
   obj_t *new;
   int classNdx;

   /* now load the objects in the scene */
   while (fscanf(in, "%128s", objclass) == 1){
      if (objclass[0] == '#') {
         /* It's a comment -- skip the rest of the line */
         fgets(buf, sizeof(buf), in); /* consume rest of line */
      }
      else {
         for (classNdx = 0; classes[classNdx].className != NULL; classNdx++) {
             if (strcasecmp(objclass, classes[classNdx].className) == 0) {
                break;
             }
         }
         /* Call the object initialization routine */
         new = classes[classNdx].classInit(in, objclass);
         if (new == NULL) {
            fprintf(stderr, "failed to load type %s \n", objclass);
            model_dump(stderr, model);
            exit(2);
         }
         else {
            list_add(model->scene, new);
         }
      } /* End else "not a comment" */
   }

   return(0);
}
```

The model_dump() routine could then be:

```
/**/
void model_dump(
FILE   *out,
model_t *model)
{
   obj_t *obj;
   int classNdx;

   obj = model->scene->head;
   while (obj != NULL)
   {
      for (classNdx = 0; classes[classNdx].className != NULL; classNdx++) {
         if (strcasecmp(obj->objclass, classes[classNdx].className) == 0) {
            break;
         }
      }
      classes[classNdx].classDump(out, obj);

      obj = obj->next;
   }
}
```

Note that with this table new objects can be added to the ray tracer by simply adding one line to the initialization of the *classes[]* array. The code in model_init() and model_dump() would not need to be changed.