The generic object structure

Even though C is technically not an Object Oriented language it is possible to employ mechanisms that emulate both the inheritance and polymorphism found in true Object Oriented languages.

**Inheritance**

The `object_t` structure serves as the generic “base class” from which specializations such as `planes` or `spheres` are derived. As such, it carries only the attributes that are common to all derived objects.

Specializations inherit the attributes of the classes from which derived. Specialized attributes of a `plane` are carried by a `plane_t` structure. Specialized attributes of a `sphere` are carried by a `sphere_t` structure. The `priv` pointer of the `object_t` provides a link to the `plane_t` or `sphere_t` and is thus declared as `void *`.

**Polymorphism**

Polymorphic behavior, in which the base `object_t` class provides a default behavior that can be overridden by specializations of the object, is achieved by the use of function pointers embedded in the `object_t` (or its subordinate specialization.) These can be initialized to point to functions that provide a default behavior but may be overridden as needed when an esoteric object such as a `tiled plane` must substitute its own “method”.
**Example of inheritance**

An inheritance hierarchy is based upon the principle of increased specialization. In its "purest" form, inheritance can be represented by a proper tree as shown below.

- The *base* class carries attributes that are common to all classes, and virtual functions that may or may not be overridden.
- Attributes that are specific to a particular entity *plane/normal* or *sphere/radius* are not defined in the *base class*.
- The derived class *inherits* the attributes of classes above it in the class hierarchy.
- The specialization can continue over multiple levels.
- The amount of “new stuff” required in the implementation of the derived class can range from trivial (*ellipsoid_t*, *fplane_t*) to moderate (*plane_t*, *sphere_t*) to fairly complex (*texplane_t*).
The generic object structure

typedef struct object_type
{
    int     cookie;
    char    obj_type[NAME_LEN]; /* entity type plane, sphere,.. */
    char    obj_name[NAME_LEN]; /* entity instance name, floor */

    /* Function pointers that can be overridden to provide polymorphic behavior */
    void    (*printer)(struct object_type *, FILE *);
    double  (*hits)(struct object_type *, vec_t *, vec_t *);
    void    (*ambient)(struct object_type *, struct material_type *, drgb_t *);
    void    (*diffuse)(struct object_type *, struct material_type *, drgb_t *);
    double  (*specular)(struct object_type *, struct material_type *);

    /* Pointer to associated material structure */
    material_t *mat;

    /* Data associated with last hit point */
    vec_t  last_hit;    /* Last hit point */
    vec_t  last_normal; /* Normal at last hit point */
    void   *priv;       /* Private type-dependent data */
} object_t;
Declaration of derived object types

The specific characteristics of derived object types must be carried by structures that are specific to the object type being described. The priv pointer of the base class object_t is used to connect the generic instance to the esoteric instance. This connection is automatic and invisible in a true OO language but is manual and visible in C.

Notice that the process of refinement or specialization can continue over multiple levels. The priv pointer of the plane structure may point to an fplane (bounded rectangular plane) refinement; or a tplane (tiled plane) refinement.

/* This structure carries the attributes */
/* of an infinite (unbounded) plane */
typedef struct plane_type
{
    vec_t normal;   /* vector perpendicular to plane */
    vec_t point;    /* any point on the plane */
    double ndotq;   /* dot product of normal and point */
    void  *priv;    /* Data for specialized types */
} plane_t;

/* Sphere */
typedef struct sphere_type
{
    vec_t center;
    double radius;
    vec_t scale;   /* for ellipsoids */
    void  *priv;
} sphere_t;
Pointers to functions

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

```c
#include <stdio.h>

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

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

    ptrf = adder;           // point it to adder (note no &)
                           // is needed (but it doesn't hurt))

    sum = (*ptrf)(3, 4);    // invoke it (*ptrf) parens req'd!
    printf("sum = %d \n", sum);
    return(0);
}

==> a.out
sum = 7
```
Function pointers as do-it-yourself polymorphism

Recall that the \textit{object\_t} structure contains function pointers:

\begin{verbatim}
typedef struct object\_type
{
    int     cookie;
    char    obj\_type[NAME\_LEN];  /* entity type plane, sphere,.. */
    char    obj\_name[NAME\_LEN];  /* entity instance name, floor */
\} object\_t;

/* Function pointers that can be overridden to provide polymorphic behavior */
void   (*printer)(struct object\_type *, FILE *);
double (*hits)(struct object\_type *, vec\_t *, vec\_t *);
void   (*ambient)(struct object\_type *, struct material\_type *, drgb\_t *);
void   (*diffuse)(struct object\_type *, struct material\_type *, drgb\_t *);
double (*specular)(struct object\_t *, struct material\_type *);
\end{verbatim}

These pointers must be set in \textit{object\_init()} to provide the default behavior. The elements on the right side of the equal sign:

- must be the names of functions having parameters that match the arguments in the above prototypes

\begin{verbatim}
obj->printer = object\_print;         // These must be functions
obj->hits    = object\_no\_hit;        // ... with matching parameters
obj->ambient = material\_getambient;
... etc.
\end{verbatim}

The \textit{plane\_init} function must override these default settings providing its own functions that implement the characteristic behavior of the plane. In this way we can emulate polymorphic behavior in the C language.

\begin{verbatim}
obj->printer = plane\_print;
obj->hits    = plane\_hits;
\end{verbatim}
Implementing polymorphic functions

The mission of a hits function is to determine if a ray fired from location base in unit direction dir hits object obj.

Needless to say a completely different strategy is required to determine if a ray intersects a plane and if it intersects a sphere.

Therefore each visible object must provide its own hit testing function and override the default function (which always returns miss).

All of the hits functions have the same parameters as the prototype in the struct object_type:

```c
double (*hits)(struct object_type *, vec_t *, vec_t *);
```

double object_no_hit(
object_t *obj, /* Candidate object */
vec_t *base, /* Start point of ray */
vec_t *dir) /* MUST be unit vector */
{
    return(-1.0); // negative distance means miss
}

double plane_hits(
object_t *obj, /* Candidate object */
vec_t *base, /* Start point of ray */
vec_t *dir) /* MUST be unit vector */
{
}

double sphere_hits(
object_t *obj, /* Candidate object */
vec_t *base, /* Start point of ray */
vec_t *dir) /* MUST be unit vector */
{
}
**Invoking a polymorphic function**

When a function pointer is contained in a structure, and an entity holds a pointer to the structure, the polymorphic function is called in the following way.

The actual arguments passed to the function must be the same in number and type as declared in the function pointer and in the actual implementation of the function.

```c
    dist = obj->hits(obj, &ray_base, &ray_dir);
```

Note that the caller of the polymorphic function does not know what actual function is being invoked.
The `object.c` module

This module contains functions used in initializing and printing the generic object.

**object_init() -**

This function performs operations analogous to `camera_init()` and `material_init()`. An object definition is shown below. The token "sphere" will be consumed prior to `object_init` being called. The `object_init()` function is responsible for consuming the data shown in red. The remainder of the attributes will be consumed by `sphere_init()`.

```
sphere   center
material  steelblue
center    4.0   1.0  -6.0
radius    5.0
```

```c
void_t object_init(FILE *in, model_t *model)
{
   object_t *obj;
   material_t *mat;
   char buf[NAME_LEN];
   int count;

   /* Create a new object structure and zero it */
   obj = malloc(sizeof(object_t));
   assert(obj != NULL);
   memset(obj, 0, sizeof(object_t));
   obj->cookie = OBJ_COOKIE;

   /* Read the descriptive name of the object */
   /* left_wall, center_sphere, etc.          */
   count = fscanf(in, "%s", obj->obj_name);
   assert(count == 1);

   /* Consume the delimiter { */
   count = fscanf(in, "%s", buf);
   assert(buf[0] == '{');

   /* The first attribute must be material */
   count = fscanf(in, "%s", buf);
   assert(count == 1);
   assert(strcmp(buf, "material") == 0);

   /* Now get the name of the material (blue, green, etc) */
   count = fscanf(in, "%s", buf);
   assert(count == 1);

   /* If the material is defined, save a pointer to the */
```
/* mat structure in the object structure. Failure to */
/* find the material is a fatal error.            */
mat = material_getbyname(model, buf);
assert(mat != NULL);

obj->mat = mat;

/* Initialize default handlers */
obj->printer = object_print;
obj->hits = object_no_hit;
obj->ambient = material_getambient;
obj->diffuse = material_getdiffuse;
obj->specular = material_getspecular;

/* Finally add the object to the list */
list_add(model->objs, (void *)obj);

}
**object_no_hit()** -

This function just returns a code that means the ray missed the object. As such it should always be overridden. It is provided to provide warning and avoid segfaults in case a developer of a new object type fails to establish a hits function.

```c
double object_no_hit(object_t *obj, /* Candidate object */
                     vec_t *base,  /* Start point of ray */
                     vec_t *dir)   /* MUST be unit vector */
{
    fprintf(stderr, "Object %s failed to provide hit func \n", obj->object_name);
    return(-1.0); // negative distance means miss
}
```

**object_print()** -

The `object_print()` function should print the `object_type` and `object_name` along with the word "material" and the name of the material. The format should be consistent with other printers.

```c
void object_print(object_t *obj, FILE *out)
{
    /* should only take care of the things in red above */
}
```
object_list_print() -

The `object_list_print()` function processes the entire object list. It should call the polymorphic function `obj->printer()` to print each object.

**NOTE:** `object_list_print()` should call the polymorphic printer `obj->printer` and NOT CALL `object_print()` directly.

```c
/* Produce a formatted dump of the material list */

void object_list_print(model_t *model, FILE *out)
{
    for each obj in the model->objs list
    {
        assert(obj->cookie == OBJ_COOKIE)
        invoke polymorphic printer method
    }
}
```
The plane.c module

The infinite plane is the simplest type of visible object. It is useful in building "floors", "ceilings", and "walls". A plane in 3-D spaces is defined by:

- The location of any point on the plane.
- A vector (called the normal) that is perpendicular to the plane

```c
typedef struct plane_type
{
    vec_t    normal;    /* read from model description */
    vec_t    point;     /* read from model description */
    double   ndotq;     /* normal dot point */
    void     *priv;     /* Data for specialized types */
} plane_t;
```
The `plane_init()` function has several responsibilities that are described below.

In a true object oriented language, the `plane_type` will be a specialization of the `object_type` and when a new instance of `plane_type` is created the constructors for BOTH the plane and object classes will be AUTOMATICALLY invoked in top down order. When simulating inheritance with C we must

- explicitly invoke the constructors for each element in the hierarchy and
- link the structures that represent them together.

In C++ this will implicitly require that generic attributes appear first in the model definition and so we will implement our pseudo-constructors in a compatible way.
The `plane_init()` function works in a way analogous to `camera_init()` and `material_init()`. One difference is that it must interact with `object_init()`.

```c
// int attrmax is maximum number of attributes
void plane_init(FILE *in, model_t *model, int attrmax)
{
    plane_t  *pln;
    object_t *obj;
    int       count;

    /* Call the object_init() function to create the */
    /* object_t and process the "material" attribute */
    /* Use list_get_entity() to make obj point to the */
    /* newly created object_t structure. */
    /* Your list_add() function must set current to the */
    /* last element in the list for this to work */
    /* correctly. */

    /* malloc a plane_t structure and set the priv */
    /* pointer in the object_t structure to point to */
    /* the plane_t */

    /* Store the word "plane" in the object_type field of */
    /* the object_t structure. Use the strcpy() function */

    /* Ask plane_load_attributes to load the attributes */
    /* Attributes are normal and point */
    count = plane_load_attributes(in, pln, attrmax);
    assert(count == 2);

    /* Set obj->hits to plane_hits() function and */
    /* obj->printer to plane_print() */

    /* pre-compute ndotq */
}
```
**plane_load_attributes() -**

This function works just like your other attribute loaders. It must return the number of attributes loaded.

```c
int plane_load_attributes(FILE *in, plane_t *pln, int attrmax)
{
    ----- like camera_load_attributes ------
}
```

**plane_print() -**

Each object specific printer is responsible for first *calling its parent's print function.*

```c
void plane_print(object_t *obj, FILE *out)
{
    plane_t *pln;

    /* Print generic attributes */
    object_print(obj, out);
    /* Recover pln pointer from object_t and print */
    /* point and normal in usual format */
}
```

Output should look like:

```
plane     floor     <-- printed by object_print()
material  gray
normal     0.0  1.0  0.0  <-- printed by plane_print()
point      0.0  -0.1  0.0
```
**Hit functions**

Given the viewpoint, ray direction, and a pointer to an `object_t`, the mission of a hit function is to determine if the ray hits the object. If it does, the hit point and the normal vector at the hit point should be stored in the `object_t`, and the distance to the hit point returned to the caller. The function should return -1.0 on a miss.

![Diagram of hit functions](image)

Given $V$, $D$, and an object structure $O$, the mission of a hit function is to determine if a ray based at $V$ traveling in direction $D$ hits $O$.

*All* points on the ray may be expressed as a function of a single parameter $t$ where $t$ is the distance along the ray from the viewpoint. Every point $P$ on the ray may thus be expressed as:

$$V + tD \quad \text{where} \quad -\infty < t < \infty \quad \text{for some } t$$

If $P$ is a point on the ray, the following relations hold:

- Distance to point: $t = \| P - V \|$
- Location of point: $P = V + tD$
**General Quadric Surfaces**

These surfaces are so named because the variables $x, y, \text{ and } z$ take on at most the power of two. The general equation for the quadric is given below:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz = J$$

We will start with two of the simpler ones:

The sphere:

$$x^2 + y^2 + z^2 = r^2$$

and its relative, the ellipsoid:

$$Ax^2 + By^2 + Cz^2 = r^2$$

The plane:

$$Gx + Hy + Iz = J$$

for the plane

- the plane normal $N$ is $(G, H, I)$ and
- $J$ is chosen so that the plane passes through the specified point $Q$.

Quadric surfaces are “nice” in a raytracing environment because the intersection of a ray with the surface may always be found by solving, at worst, a quadratic equation.
Determining if a ray hits a plane

plane_hits() -

The plane_hits() function determines if a ray hits a plane and if so, fills in the coordinates of the hitpoint and the normal in the object_t structure and returns the distance from the base to the point of contact.

double plane_hits(
    object_t  *obj,
    vec_t     *base,      /* ray base              */
    vec_t     *dir)       /* unit direction vector */
{

    /* The steps for the algorithm are numbered with the green numbers below */
}

This basic strategy will be used in all hits functions:
0 - Assume that \( V \) represents the start of the ray and \( D \) is a unit vector in its direction
1 - Derive an equation for an arbitrary point \( P \) on the surface of the object.
2 - Recall that all points on the ray are expressed as \( V + tD \)
3 - Substitute \( V + tD \) for \( P \) in the equation derived in (1).
4 - Attempt to solve the equation for \( t \).
5 - If a solution \( t_h \) can be found, then \( H = V + t_h D \).

A plane in three dimensional space is defined by two parameters

A normal vector \( N = (n_x, n_y, n_z) \)
A point \( Q = (q_x, q_y, q_z) \) through which the plane passes.

A point \( P = (p_x, p_y, p_z) \) is on the plane if and only if:
\[ N \text{ dot } (P - Q) = 0 \] because, if the two points \( P, Q \) lie in the plane, then the vector from one to the other \( (P - Q) \) also lies in the plane and thus it is necessarily perpendicular to the plane's normal.

We can rearrange this expression to get:
\[ N \text{ dot } P - N \text{ dot } Q = 0 \]
\[ N \text{ dot } P = N \text{ dot } Q \] \hspace{1cm} (1)

Note that in this equation \( N \) and \( Q \) are known attributes of the plane and \( P \) is the unknown. Recall that the location of any points on a ray based at \( V \) with direction \( D \) is given by:
\[ V + t D \]

Therefore we may replace the \( P \) in equation (1) by \( V + tD \) and get:
\[ N \text{ dot } (V + tD) = N \text{ dot } Q \] \hspace{1cm} (2)
Some algebraic simplification allows us to solve this for $t$

$$
N \cdot (V + tD) = N \cdot Q \quad (2)
$$

$$
N \cdot V + N \cdot tD = N \cdot Q
$$

$$
N \cdot tD = N \cdot Q - N \cdot V
$$

$$
t (N \cdot D) = (N \cdot Q - N \cdot V) \quad (3)
$$

You can see that three dot products will need to be done in order to compute the distance, $t_h$. So, start with those before doing the above formula for distance:

$$
N \cdot Q
$$

$$
N \cdot V
$$

$$
N \cdot D
$$

#1 Since $N \cdot D$ is in the denominator of the above formula for computing distance from $(3)$ above, you can get that value first and check to see if it is zero or not. If it is zero, then that means the direction of the ray is perpendicular to the normal to the plane; which means the ray is parallel to the plane. Either the ray lies in the plane or misses the plane entirely. We will always consider this case a miss and return -1. Attempting to divide by 0 will cause your program to either fault and die or return a meaningless value.

#2 Get the other dot product values: $N \cdot V$ and $N \cdot Q$. Then plug them into the distance formula above.

Unlike other quadric surfaces, there is only a single point at which a ray intercepts a plane. Therefore, unlike equations we will see later, this one is not quadratic. There are some other special cases we must consider besides the one mentioned above where the ray is parallel to the plane:

#4 The first one is if $t_h < 0$. In this case, the hit lies behind the viewpoint rather than in the direction of the screen. This should be considered a miss and -1 should be returned.

The second one is if hit point lies on the viewpoint side of the screen. Compute the hit point first:

#5 $H = V + t_hD$

#6 With that hit point $H$, which is a vector $(h_x, h_y, h_z)$, if $h_z > 0$ - the hit is on the wrong side and -1 should be returned.
The *location of the hitpoint* that should be stored in the `object_t` if -1 was not returned, is that $H$ from step #5 above:

$$ H = V + t_hD $$

The *normal at the hitpoint* which must also be saved in the `object_t` is just $N$

Don't forget the very last thing to do in the function is to return the value for $t_h$, which is the distance - - #3 from the previous page.
Testing the object, material and plane modules

At this point, it is useful to extend our material tester to include loading and testing a plane object. Each time we add a new component to the project we want to ensure that we haven't broken any existing component. This process is called regression testing.

We will use the following input. The first plane is sometimes called a "back wall". All of its points have a z-coordinate of -7 and its normal points directly outward in the +z direction. The second plane is called a "floor" but a normal floor would have a normal of 0 1 0. This "floor" is "tilted" down toward the viewer. Elements of the floor with y coordinates > 0.0 will be hidden behind the back wall.

A side view of the situation is shown below.

```
material green
{
    ambient 0 5 0
}

material brown
{
    ambient 3 3 0
}

plane wall
{
    material green
    normal 0 0 1
    point 0 0 -7
}

plane floor
{
    material brown
    normal 0 1 1
    point 0 0 -7
}
```
/* plntest.c */

#include "ray.h"

int main(int argc, char *argv[]) {
    model_t    mod;
    model_t    *model = &mod;
    material_t *mat;
    object_t   *obj1;
    object_t   *obj2;
    char       entity[16];
    int        count;
    FILE *infile;
    FILE *outfile;
    assert(argc > 3);
    infile = fopen(argv[1], "r");
    assert(infile != NULL);
    outfile = fopen(argc[2], "w");
    assert(outfile != NULL);

    /* Create lists */
    model->mats = list_init();
    model->objs = list_init();

    /* Load the two material definitions */
    count = fscanf(infile, "%s", entity);
    material_init(infile, model, 0);
    mat = (material_t *)list_get_entity(model->mats);
    assert(mat->cookie == MAT_COOKIE);
    fprintf(stderr, "loaded %s \n", mat->name);

    count = fscanf(infile, "%s", entity);
    material_init(infile, model, 0);
    mat = (material_t *)list_get_entity(model->mats);
    assert(mat->cookie == MAT_COOKIE);
    fprintf(stderr, "loaded %s \n", mat->name);

    /* Verify that worked */
    material_list_print(model, stderr);

    /* Now load the two object definitions */
    count = fscanf(infile, "%s", entity);
    plane_init(infile, model, 0);
    obj1 = (object_t *)list_get_entity(model->objs);
    assert(obj1->cookie == OBJ_COOKIE);
    fprintf(stderr, "loaded %s \n", obj1->obj_name);

    object_list_print(model, stderr);

    count = fscanf(infile, "%s", entity);
    plane_init(infile, model, 0);
    obj2 = (object_t *)list_get_entity(model->objs);
    assert(obj2->cookie == OBJ_COOKIE);
    fprintf(stderr, "loaded %s \n", obj2->obj_name);

    /* Verify that worked */
    object_list_print(model, stderr);
}
Now test the hits functions

```c
vec_t view = {4.0, 3.0, 5.0};
vec_t dir = {0.0, 0.0, -1.0};
double dist = 0.0;
vec_t unit;

vec_unit(&dir, &unit);
memset(obj1->hits, 0, sizeof(vec_t));
dist = obj1->hits(obj1, &view, &unit);
fprintf(stderr, "dist to plane 1 %8.3lf \n", dist);
vec_print(stderr, "hit point", obj1->last_hit);

vec_unit(&dir, &unit);
memset(obj2->hits, 0, sizeof(vec_t));
dist = obj2->hits(obj2, &view, &unit);
fprintf(stderr, "dist to plane 2 %8.3lf \n", dist);
vec_print(stderr, "hit point", obj2->last_hit);

/* Make sure we don't get a hit in +z space */
dir.y = -2.1;
vec_unit(&dir, &unit);
memset(obj2->hits, 0, sizeof(vec_t));
dist = obj2->hits(obj2, &view, &unit);
fprintf(stderr, "positive z test \n");
fprintf(stderr, "dist to plane 2 %8.3lf \n", dist);
vec_print(stderr, "hit point", obj2->last_hit);

/* Make sure we don't get a hit on a miss... Shoot straight down at back wall */
dir.z = 0;
vec_unit(&dir, &unit);
memset(obj1->hits, 0, sizeof(vec_t));
dist = obj1->hits(obj1, &view, &unit);
fprintf(stderr, "vertical ray test \n");
fprintf(stderr, "dist to plane 1 %8.3lf \n", dist);
vec_print(stderr, "hit point", obj1->last_hit);

return(0);
```

24
Program output

loaded green
loaded brown
material   green
ambient   0.0   5.0   0.0
diffuse   0.0   0.0   0.0
specular  0.0   0.0   0.0

material   brown
ambient   3.0   3.0   0.0
diffuse   0.0   0.0   0.0
specular  0.0   0.0   0.0

loaded wall

plane     wall
material   green
normal    0.0   0.0   1.0
point     0.0   0.0  -7.0
loaded floor

plane     wall
material   green
normal    0.0   0.0   1.0
point     0.0   0.0  -7.0

plane     floor
material   brown
normal    0.0   1.0   1.0
point     0.0   0.0  -7.0

dist to plane 1   12.000
hit point   4.000   3.000  -7.000

dist to plane 2   15.000
hit point   4.000   3.000 -10.000

positive z test
dist to plane 2   -1.000
hit point   4.000  -7.161   0.161

vertical ray test
dist to plane 1   -1.000
hit point   4.000   3.000  -7.000
The sphere.c module

A sphere in 3-D space is defined by:

- The location of its center, a vec_t.
- The radius of the sphere which is a scalar (double) value, \( r \).

```c
typedef struct sphere_type
{
    vec_t center;
    double radius;
    vec_t scale;  // (1, 1, 1) for spheres
} sphere_t;
```

**sphere_init()**

The `sphere_init()` function performs the same functions as `plane_init()`.

```c
int sphere_init(FILE *in, model_t *model, int attrmax)
{
    sphere_t *sph;
    object_t *obj;
    int count;

    /* call object_init() function to create the object_t 
     * and process the "material" attribute */

    /* use list_get_entity() to make obj point to the newly 
     * created object_t structure. */

    /* malloc a sphere_t structure and set the priv pointer 
     * in the object_t structure to point to the sphere_t */

    /* use strcpy() to store the word "sphere" to the 
     * object_type field of the object_t structure */

    return 0;
}
```
/* ask sphere_load_attributes() to load the attributes which are center and radius */

    count = sphere_load_attributes(in, sph, attrmax);
    assert(count == 2);

/* set up the polymorphic function pointers in object_t
set obj->hits to sphere_hits() function
set obj->printer to sphere_print() function */

}

sphere_load_attributes() -

This function works just like your other attribute loaders. It must return the number of attributes loaded.

int sphere_load_attributes(FILE *in, sphere_t *sph, int attrmax)
{

    ----- like other load attributes functions ----- 

}

sphere_print() -

Each object specific printer is responsible for first calling its parent's print function.

void sphere_print(object_t *obj, FILE *out)
{

    ---- print generic attributes first (call parent's print function)
    object_print(obj, out);

    ---- then print sphere's attributes (format like other print functions)
    ---- how to do access the sphere's attributes??

}

Output should look like

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sphere</td>
<td>bigball</td>
<td></td>
</tr>
<tr>
<td>material</td>
<td>green</td>
<td></td>
</tr>
<tr>
<td>center</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.4</td>
<td>-2.0</td>
</tr>
</tbody>
</table>

<- printed by object_print()
<- printed by sphere_print()
Determining if a ray hits a sphere.

*sphere_hits()* -

The *sphere_hits()* function determines if a ray hits a sphere and if so fills in the coordinates of the *hitpoint* and the *normal* in the *object_t* structure.

```c
double sphere_hits(
    object_t *obj,
    vec_t *base,    /* ray base */
    vec_t *dir)     /* unit direction vector */
{
    /* The steps for the algorithm are numbered with the green numbers below */
}
```

Assume the following:

- \(V\) = viewpoint or start of the ray
- \(D\) = a unit vector in the direction the ray is traveling
- \(C\) = center of the sphere
- \(r\) = radius of the sphere

The arithmetic is much simpler if the center of the sphere is at the origin. So we start by moving it there! To do so we must make a compensating adjustment to the base of the ray.

\[
C' = C - C = (0, 0, 0) = \text{new center of sphere}
\]

\#1  \(V' = V - C = \text{new base of ray}\)

\(D\) does not change
A point $P$ on the sphere whose center is $(0, 0, 0)$ necessarily satisfies the following equation:

$$p_x^2 + p_y^2 + p_z^2 = r^2 \quad (1)$$

All points on the ray may be expressed in the form

$$P = V' + t D = (v'_x + td_x, v'_y + td_y, v'_z + td_z) \quad (2)$$

where $t$ is the Euclidean distance from $V'$ to $P$.

Thus we need to find a value of $t$ which yields a point that satisfies the two equations. To do that we take the $(x, y, z)$ coordinates from equation (2) and plug them into equation (1). We will show that this leads to a quadratic equation in $t$ which can be solved via the quadratic formula.

$$(v'_x + td_x)^2 + (v'_y + td_y)^2 + (v'_z + td_z)^2 = r^2$$
Expanding this expression

\[(v'_x + td_x)^2 + (v'_y + td_y)^2 + (v'_z + td_z)^2 = r^2\]

by squaring the three binomials yields:

\[
(v'_x^2 + 2tv'_x d_x + t^2 d_x^2) + (v'_y^2 + 2tv'_y d_y + t^2 d_y^2) +
(v'_z^2 + 2tv'_z d_z + t^2 d_z^2) = r^2
\]

Next we collect the terms associated with common powers of \(t\)

\[
(v'_x^2 + v'_y^2 + v'_z^2) + 2t (v'_x d_x + v'_y d_y + v'_z d_z) +
t^2(d_x^2 + d_y^2 + d_z^2) = r^2
\]

Now we reorder terms as decreasing powers of \(t\) and note that all three of the parenthesized tri-nomials represent dot products.

\[
(D \cdot D) t^2 + 2 (V' \cdot D) t + V' \cdot V' - r^2 = 0
\]

We now make the notational changes:

\[\#2 \quad a = D \cdot D\]

\[\#3 \quad b = 2 (V' \cdot D)\]

\[\#4 \quad c = V' \cdot V' - r^2\]

to obtain the following equation

\[at^2 + bt + c = 0\]

whose solution is the standard form of the quadratic formula:

\[t_h = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\]
Recall that quadratic equations may have 0, 1, or 2 real roots depending upon whether the discriminant:

\[ (b^2 - 4ac) \]

is negative, zero, or positive. These three cases have the following physical implications:

- **negative** => ray doesn't hit the sphere – return -1
- **zero** => ray is tangent to the sphere hitting it at one point – we will consider this a miss – return -1
- **positive** => ray does hit the sphere and would pass through its interior – this is the *only* case we consider a hit; can continue with computing distance if haven't returned yet.

Furthermore, the two values of \( t \) are the distances from the base of the ray to the points(s) of contact with the sphere. We always seek the *smaller* of the two values since we seek to find the “entry wound” not the “exit wound”.

Therefore, the `sphere_hits()` function should return the following if the discriminant is positive:

\[ t_h = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \]

otherwise, it should have returned -1 from step **6** above.
Determining the coordinates of the hit point on a sphere.

The \((x, y, z)\) coordinates are computed as follows.

\[ H = V + t_h D \]

where \( t_h \) is the smaller root of the quadratic equation on the previous page.

After computing the hitpoint, save (copy) it to the object's last_hit field.

Important items to note are:
- The actual base of the ray \( V \) and not the translated base \( V' \) must be used.
- The vector \( D \) must be a unit vector in the direction of the ray.

Determining the surface normal at the hit point.

The normal at any point \( P \) on the surface of a sphere is a vector from the center to the point. Thus

\[ N = P - C \quad (\text{note that } N \text{ will be a unit vector } \iff r = 1) \]

Therefore a unit normal may be constructed as follows:

\[ N_u = (H - C) / \| (H - C) \| \]

After computing the unit normal, save (copy) it to the object's last_normal field.

Returning the distance.

The very last thing to do in the function is to return the value for \( t_h \), which is the distance - - from previous page