An object oriented list structure

This approach is the C++ version of the *external* list structure we developed in C. It provides a reasonably simple mechanism for us to consider various aspects of C++.

Advantages include:

- The `link_t` objects remain invisible to the outside world.

- Facilitating the use of multiple lists of different kinds of entities. For example, with this technique it is easy to create a list of lights, materials, or visible objects *without modifying the object structure or the list structure.*

Disadvantages include:

- How to safely process the list when a new function is called by a function already in the middle of processing the list.

```c
list_t {
  link_t *first;
  link_t *last;
  link_t *current;
}

link_t {
  link_t *next;
  void *entity;
};

object_t{
}
```
Example linked list classes

The link and list classes are correctly declared below: Note that the default constructor is explicitly provided and that it is overloaded.

A C++ class extension to the C structure.

- The class definitions should go in a file called `list.h`.
- The actual implementations of the functions should go in a file called `list.cpp`

Recall that it was necessary to use `typedef` to create a name called `object_t` that was equivalent to `struct object_type`. In C++, the class declaration implicitly creates the user defined type `link_t` shown below.

Each class has a special function called the constructor. Its name is the same as the name of the class and it is automatically invoked each time a new instance of the class is created. Constructors are never allowed to return a value. A class can provide multiple constructors. The one actually invoked is the one that matches the parameters supplied when the instance is created. Other function prototypes included in the class definition are called “class methods”.

- Public elements of a class may be accessed by any entity that holds a pointer to the class instance.
- Private elements may only be accessed by class methods.

Each class also has a destructor. This function is invoked when an instance of the class is deleted.

```cpp
class link_t
{
    public:
        link_t(void);                // constructor
        link_t(void *);              // constructor
        ~link_t(void);               // destructor
        void     set_next(link_t *);
        link_t  *get_next(void);
        void    *get_entity(void);

    private:
        link_t  *next;
        void    *entity;
};
```
The list_t class.

We will build a list class that has the same functionality as our C list module. Class methods are typically invoked via pointers to the instance of the class (e.g., list->reset()). For this reason, some programmers tend to discard the redundant part of the method name when transitioning from C and use simply reset instead of list_reset which would then have to be invoked as list->list_reset().

We will use both approaches.

class list_t
{
public:
    list_t(void); // constructor
    list_t(const list_t &); // copy constructor
    ~list_t(void); // destructor
    void add(void *entity); // add entity to end of list
    void reset(void); // set current to start of list
    void *get_entity(void); // get current entity in list
    int not_end(void);
    void next_link(void);
private:
    link_t *first;
    link_t *last;
    link_t *current;
};
link_t class methods

The link_t and list_t class methods will reside in list.cpp.

This link_t constructor is passed a pointer to the entity which this new link will own. Its mission is to set the next pointer to NULL and the entity pointer to the new entity being added to the list. Notice that it is not necessary to malloc the new link_t. That is done "automagically" within the new mechanism.

The :: operator is called the scope operator. It is used to declare that the link_t( ) constructor function belongs to the link_t class. It is always necessary to use the scope operator when:

● prototypes are used in the class declaration and
● the function body is defined outside the class declaration.

NOTE WELL: Class methods are always invoked in the context of an instance of the class. Therefore, data elements of the class must be DIRECTLY accessed as shown below.

```
link_t::link_t(void *newentity)
{
    next = NULL;
    entity = newentity;
}
```

Programmers transitioning from C often experience an intense need to write the above code incorrectly as shown below. Try to avoid this temptation. It will NEVER work.

```
link_t::link_t(void *newentity)
{
    link->next = NULL;
    link->entity = newentity;
}
```
The `set_next()` method is a typical “setter” function that is used to tell the `link_t` to manipulate its own `next` pointer. It is called by the `add` method of the `list_t` class when an item that is not the first item is added to the list. It should set the `next` attribute of the `link_t` to `new_next`:

```cpp
void link_t::set_next(link_t *new_next)
{ /* set the next element of this link to new_next */
}
```

The `get_next()` method is a typical “getter” function that is used as a way to tell the `link_t` to cough up the value of its own `next` pointer:

```cpp
link_t * link_t::get_next()
{ /* return the next pointer of this link */
}
```

The `get_entity()` method is analogous. It would also work to simply make all of the `next` and `entity` elements `public`. Then any holder of a reference to the `link_t` could simply manipulate them directly... but it would be a violation of OO dogma to do so:

```cpp
void * link_t::get_entity()
{ /* return the entity pointer of this link */
}
```
list_t class methods

The list_t class overrides the default constructor with its own constructor with no parameters:

```cpp
list_t::list_t()
{
    /* Set first, last, and current pointers to NULL */
}
```

Adding a new entity to the list

The `add()` method creates a new `link_t` and passes its constructor a pointer to the entity. The `new` operator shown below will create a new instance of the `link_t`. Since a `void *` parameter is passed, the correct constructor will be called.

```cpp
void list_t::add(void *entity)
{
    /* Create a new link passing it the entity pointer */

    link_t *link;
    link = new link_t(entity);

    /* Now add the link to the list using generally the */
    /* same approach as the C version */
}
```

Resetting the current pointer to the first element of the list

The `reset()` method sets the current pointer to the first element in the list and returns a pointer to the first entity in the list.

```cpp
void * list_t::reset(void)
{
    /* Set the current link pointer to the first pointer. */
}
```

Retrieving the entity associated with the current list element

The `get_entity()` method returns the entity pointer associated with the current link.

```cpp
void * list_t::get_entity(void)
{
    assert(current != NULL);

    /* Use the get_entity method of the link_t */
}
```
Testing to see if the current pointer is at the end.

This function works just like it did in the C version.

```cpp
int list_t::not_end(void)
{
    /* If the current point is NULL return(0), otherwise */
    /* return(1) */
}
```

Advancing the current pointer

Use the get_next() method of the link_t class to update current.

```cpp
void list_t::next_link(void)
{
    assert(current != NULL);
    current = current->get_next();
}
```

Deleting the list

The delete() operator can be used to free class instances created with new. However, the instances of the link_t class are not automatically deleted. Therefore, it is necessary to create a list_t destructor that processes all the links deleting them one at a time.

```cpp
list_t::~list_t()
{
    current = first;
    link_t *kill;
    fprintf(stderr, "in list destructor \n");
    /* For each link in the set kill to current, update current */
    /* and delete kill */
}
```
Creating a list

This code from listmain.c illustrates how to use constructors to create new instances of a class e_t, create a new list and add elements to a list. The call to scanf() works only because e_name and e_id have the public protection attribute.

```cpp
class e_t {
    public:
        char e_name[NAME_LEN];    // entity name
        int  e_id;                // entity id code
};

int main(int argc, char *argv[]) {
    list_t *elist;
    e_t    *eloc;
    char   name[NAME_LEN];
    int    count;

    FILE *in;
    FILE *out;

    assert(argc >= 3);
    in = fopen(argv[1], "r");
    assert(in != NULL);
    out = fopen(argv[2], "w");
    assert(out != NULL);

    /* Create new list */
    elist = new list_t;

    /* Read input file consisting of names and id codes
    adding entities to list .. This would not work if e_name and e_id were private */
    while (fscanf(in, "%s %d", name, &count) == 2) {
        eloc = new e_t;
        strcpy(eloc->e_name, name);
        eloc->e_id = count;
        elist->add((void *)eloc);
    }

    Processing a list

    The processing proceeds much like it did in C but with calls that previously looked like list_function_name now looking like elist->function_name

    /* Now play it back */
    elist->reset();
    while (elist->not_end()) {
        eloc = (e_t *)elist->get_entity();
        fprintf(out, "%s %d 
", eloc->e_name, eloc->e_id);
        elist->next_link();
    }
}
```
Deleting a list

Deleting a list of generic entities is somewhat more painful than it was in C. In C, the `free()` function can be used to release any dynamic memory regardless of structure type. In C++ the `delete` operator is used to free classes created with `new`. However, the `delete` operator must be aware of the type of class it is deleting.

Therefore it is necessary to process the list deleting the entities before deleting the list itself.

```cpp
/* Now free all list control structures and the */
/* e_t structures as well. */
elist->reset();
while (elist->not_end())
{
    eloc = (e_t *)elist->get_entity();
    delete eloc;
    elist->next_link();
}

/* Now delete the list and link structures */
delete elist;
fprintf(out, "done\n");
```
Another example class definition

A class definition creates a type name that can be used in a standalone fashion (like a *typedef*) but the explicit *typedef* is no longer required. These should replace the comparable structure definitions in *ray.h*.

```cpp
class camera_t
{
    public:
        camera_t();
        camera_t(FILE *in);
        void getdir(int x, int y, vec_t *dir);
        void store_pixel(int x, int y, drgb_t* pix);
        int getxdim(void);
        int getydim(void);
        void getviewpt(vec_t *view);
        void camera_print(FILE *out);
        void camera_write_image(FILE *out);
        void getpixsize(double *x, double *y);
    
    private:
        int  cookie;
        char name[NAME_LEN];  /* Projection screen size in pix */
        int  pixel_dim[2];    /* Screen size in world coods */
        double world_dim[2];  /* Viewpt Loc in world coods */
        vec_t view_point;     /* Build image here */
        irgb_t *pixmap;       /* Build image here */
};
```

Function prototypes that appear within a class definition are called *class methods* and are always invoked within the context of an instance of the class. They have *unqualified access to the data structures in the class*. That is, a class method will never refer to `cam->cookie`. It will simply do

```cpp
cookie = CAM_COOKIE;
```

with the understanding that the particular `cookie` is the one that belongs to *this* instance of the class.
Creating an instance of a new class:

In the C language version of the ray tracer, a new camera structure was created with a call to `camera_init()`. The `camera_init()` function then proceeded to `malloc()` a `camera_t` structure and then initialize it.

```c
camera_init(stdin, model, 0);
```

Pointers to classes are declared just as pointers to structures are. However, dynamic creation of a new class uses the `new` operator. In the C++ version of the code, we will clean things up a bit and not pass the `model` or `attrmax` parameters.

```c
camera_t *cam;
/* Load and print camera data */
cam = new camera_t(stdin);
assert(cam != NULL);
```

The `new` operator may be viewed as somewhat similar to `malloc()` but it creates an instance of the class before invoking the initializer code (which is referred to as the constructor).

Alternate ways to create an instance of a class

It is also possible to simply declare an instance of a new class and invoke its constructor:

```c
camera_t camera(stdin);
```

or

```c
camera_t camera;
```

The parameters (or lack thereof) determine which constructor is invoked. The default constructor which would be invoked in the second case doesn't actually do anything.

If parameters are supplied that don't match any prototype in the class definition, the C++ compiler responds with an appropriate nastygram. This is a very common type of error. You need to be able to recognize and correct it on your own.

```c
camera_t cam(stdin, model);
cat main.err
main.c: In function `int main(int, char**'):
main.c:14: error: no matching function for call to `camera_t::camera_t(_IO_FILE*&, model_t*)'
```
Constructors

1. Are automatically called whenever an instance of the class is created
2. Must never have a return type --- not even void.
3. May be overloaded. The function actually invoked is the one whose formal parameters “best” match the actual arguments. Thus when new camera_t(stdin) is invoked this constructor is called.

```c
camera_t::camera_t(FILE *in)
{
    char  buf[256];
    int   mask;
    assert (fscanf(in, "%s", name) == 1);
    fscanf(in, "%s", buf);
    assert(buf[0] == '{');
    cookie = CAM_COOKIE;

    camera_parse[0].loc = &pixel_dim;
    camera_parse[1].loc = &world_dim;
    camera_parse[2].loc = &view_point;

    mask = parser(in, camera_parse, NUM_ATTRS, 0);
    assert(mask == 7);

/* Allocate a pixmap to hold the ppm image data */
pixmap = (irgb_t *)malloc(sizeof(irgb_t) * pixel_dim[0] * pixel_dim[1]);
}
```

As previously noted, class data members cannot be accessed using the cam-> prefix. The major part of a C to C++ conversion is eliminating these pointer based references.

Although it is possible to implement function bodies inside class definitions, unless the functions are trivially short it leads to a big mess. The scope operator :: is saying this function belongs to the camera_t class.
**Getter and setter functions**

One objective of the O-O approach is to *encapsulate* data within instances of classes and thus protect against the uncontrolled access that is possible in standard C. Because of this, it is common for a class to export a collection of *getters* and *setters* that allow other classes access to required data but in a controlled way. For example, the *image_create()* mechanism needs the *pixel_dimension* to determine how many times it should iterate on x and y.

The implementation of class methods must employ the structure:

```cpp
return-type  class_type::function_name(parameters)
```

```cpp
int camera_t::getydim(void)  
{  
  return(pixel_dim[1]);  
}
```

A C++ program can contain a mix of C++ class methods and traditional C functions. In fact the *main()* function will always be a traditional C function. A C++ religious warrior may assert that a *real* C++ programmer will never use any other traditional C functions!

```cpp
void make_row(model_t *model, int y)  
{  
  int x;  
  int xdim;  
  camera_t *cam = model->cam;  
  xdim = cam->getxdim();  
  for (x = 0; x < xdim; x++)  
  {  
    
  }
```

**A C++ setter function**

As before, the elements belonging to the class are not qualified with a *cam-*

```cpp
void camera_t::store_pixel(int x, int y, drgb_t *pix)  
{  
  int maprow;  
  irgb_t *maploc;  
  maprow = pixel_dim[1] - y - 1;  
  maploc = pixmap + maprow * pixel_dim[0] + x;  
  scale_and_clamp(pix);  
  maploc->r = (unsigned char)pix->r;  
  maploc->g = (unsigned char)pix->g;  
  maploc->b = (unsigned char)pix->b;  
}
```
Implementing an *ad hoc* parser

The ad hoc parsers we used in the C code are readily ported to C++ as well. If you choose to use this approach you must add the class method to the class definition in ray.h.

```c
int camera_t::camera_load_attributes(FILE *in)
{
    char attrib_name[16];
    int count = 0;
    int attrcount = 0;

    /* Now consume attributes */
    count = 0;
    fscanf(in, "%s", attrib_name);
    while (attrib_name[0] != '}' && attrib_name[0] != '}
    {
        if (strcmp(attrib_name, "pixeldim") == 0) {
            count = fscanf(in, "%d %d", &pixel_dim[X], &pixel_dim[Y]);
            assert(count == 2);
            attrcount += 1;
        }
        etc....
    }
    return(attrcount);
}
```
An upside down class

Although it is common for data elements to be private and functions be public, it is sometimes useful to turn the model upside down in a hybrid C/C++ program. Here, the internal procedures do not need to support external reference (pointer) holders. It would be feasible to provide a bunch of getters that would return the pointer to the cam and the lists, but a reasonable person might conclude it is not necessary.

In general it is a good idea to make everything private that doesn't need to be public.

class model_t
{
    public:
        model_t(FILE *);
        void print(FILE *);
        camera_t *cam;
        list_t *mats;
        list_t *objs;
        list_t *lgts;

    private:
        void model_load_entity(FILE *, char *);
        void model_load(FILE *);
};
The model_t constructor

As before, note that (1) there is no need to malloc() the model structure, (2) references to class member data elements (mats, lgts, objs) are unqualified and (3) so are references to class methods (model_load())

```c
/* Init model data */
model_t::model_t(FILE *in)
{
    mats = new list_t;
    assert(mats != NULL);

    objs = new list_t;
    assert(objs != NULL);

    lgts = new list_t;
    assert(lgts != NULL);

    model_load(in);
}
```

The model_load() method

This function consumes a single entity name (material, plane, sphere, etc) and then invokes yet another class method to perform the actual loading. Note that even though this is a C++ method, in the model.cpp module, it's perfectly legal to use standard C library functions such as fscanf(), strcmp(), etc.

Some people prefer one over the other. Expect abuse from C++ religious zealots if you employ standard C I/O operations.

```c
/* Load model data */
void model_t::model_load(FILE *in)
{
    char entity[16];
    int  count;

    memset(entity, 0, sizeof(entity));

    count = fscanf(in, "%s", entity);

    /* Here entity should be one of "material",    */
    /* "light", "plane"                            */
    while (count == 1)
    {
        model_load_entity(in, entity);
        count = fscanf(in, "%s", entity);
    }
}
```
Creating new entities

Each invocation of `model_load_entity()` creates a new instance of a `material`, `plane`, `sphere`, etc.

In the C language version this is readily done using a table of function pointers. In the C++ version, a `switch` or `if else` mechanism must be used. The problem is that there is no way (that I know of) to pass a variable value to the `new` operator.

```c
new plane_t(in, this, 0);
```

If we could replace `plane_t` by a variable which could take on the value `plane_t` or `sphere_t` or `material_t` then it would be straightforward to put a table driven approach back in place.

```c
void model_t::model_load_entity(FILE *in, char *entity)
{
    if (strcmp(entity, "camera") == 0)
        cam = new camera_t(in);
    else if (strcmp(entity, "material") == 0)
        new material_t(in, this, 0);
    else if (strcmp(entity, "plane") == 0)
        new plane_t(in, this, 0);
    :
    :
    else
    {
        fprintf(stderr, "bad entity %s \n", entity);
        exit(1);
    }
}
```

Recall that the initializers for `materials and planes` needed a pointer to the `model_t` structure in order to access the lists. When such a scenario arises, the predefined `this` variable can be used.
Revisiting the model_t constructor.

In the code shown below we demonstrate that

- the this pointer can be used to provide qualified access to class data members.
- the this pointer can be copied to other pointers to provide a more C like representation

Both of these techniques are likely to evoke derision from "real" C++ programmers.

```cpp
model_t::model_t(FILE *in)
{
    model_t *model = this;

    this->mats = new list_t;
    assert(mats != NULL);

    objs = new list_t;
    assert(objs != NULL);

    model->lgts = new list_t;
    assert(lgts != NULL);

    model_load(in);
}
```
The *material_t* class

The module *material.c* is a hybrid module involving *both standard C procedures and C++ class methods*. The class methods are, for the most part, typical *getters* providing access to private values.

```cpp
class material_t
{
   public:
      material_t(){};
      material_t(FILE *in, model_t *model, int attrmax);
      void  material_getambient(drgb_t *dest);
      void  material_getdiffuse(drgb_t *dest);
      void  material_getspecular(double *spec);
      void  material_getshine(double *shiny);
      void  material_gettrans(double *trans);
      char  *material_getname();
      void  material_print(FILE *out);

   private:
      int    cookie;
      char   name[NAME_LEN];
      drgb_t ambient;      /* Reflectivity for materials */
      drgb_t diffuse;
      double specular;
      double shininess;
      double transparency;
};
```

**Standard C functions in *material.cpp***

The standard C functions are those in which the entire list of materials must be processed. Since a class method is always invoked within the context of a single instance of the class (i.e. a single material) it is not possible to make such functions members of the class. These prototypes *can not* appear inside the class definition and should remain in *rayhdrs.h*.

```c
/* Produce a formatted print of the material list */
void material_list_print(FILE *out, list_t *list)

/* Try to locate a material by name */
material_t *material_getbyname(list_t *matlist, char *name);
```
Initializing a new `material_t`

The only difference between the C and C++ version is the use of unqualified data names: `cookie, ambient, diffuse`

```c
/* Create a new material description */
material_t::material_t(FILE *in, model_t *model, int attrmax)
{
    char attrname[NAME_LEN];
    int count;
    int mask;

    /* Create a new material structure and initialize it */
    cookie = MAT_COOKIE;

    /* Read the descriptive name of the material */
    /* (dark_red, light_blue, etc.) */
    count = fscanf(in, "%s", name);
    assert(count == 1);

    count = fscanf(in, "%s", attrname);
    assert(*attrname == '{');

    mat_parse[0].loc = &ambient;
    mat_parse[1].loc = &diffuse;
    mat_parse[2].loc = &specular;
    mask = parser(in, mat_parse, NUM_ATTRS, 0);

    assert(mask != 0);
    model->mats->add((void *)this);
}
```
Interaction of C and C++ components

The mission of the *material_getbyname()* function is to find an instance of a material_t that has the proper name. As such, it may have to look at all material_t objects. Therefore it is NOT a material_t class method! Since it is not a class method, it cannot directly access the name attribute which is private.

It asks the class method *material_getname()* to return the address of the name.

Serious O-O enthusiasts *would not approve of this technique*. Returning the address of the private attribute gives the caller the ability to modify the attribute via the returned pointer!

```
/* Try to locate a material by name */
material_t *material_getbyname(list_t *list, char *name)
{
    material_t *mat;
    list->reset();
    while (list->not_end()) {
        mat = (material_t *)list->get_entity();
        if (strcmp(name, mat->material_getname()) == 0)
            return(mat);
        list->next_link();
    }
    return(NULL);
}
```

This version of the function attempts to access mat->name directly. Since name is private, it does not succeed.

```
/* Try to locate a material by name */
material_t *material_getbyname(list_t *list, char *name)
{
    material_t *mat;
    list->reset();
    while (list->not_end()) {
        mat = (material_t *)list->get_entity();
        if (strcmp(name, mat->name) == 0)
            return(mat);
        list->next_link();
    }
    return(NULL);
}
```

Since this function is *not* a class method it cannot touch the private mat->name.
The *friend* qualifier

Because it is very common in hybrid C/C++ environments for “helper” functions to need access to private elements of classes, the C++ language provides the *friend* capability.

By using *friend*, one class can give to any functions or class the right to access its private elements directly. Almost *any* use of this facility is frowned upon by O-O purists. Excessive use is frowned upon by everyone.

Nevertheless if we include the following *friend* declaration in *material_t* the error shown on the previous page will go away!

class material_t
{
    friend material_t* material_getbyname(list_t*, char*);

    public:
    material_t();
    material_t(FILE *in, model_t *model, int attrmax);
    void material_getambient(drgb_t *dest);
    void material_getdiffuse(drgb_t *dest);
    void material_getspecular(double *spec);
    void material_getshine(double *shiny);
    void material_gettrans(double *trans);
    char *material_getname();
    void material_print(FILE *out);

    private:
    int cookie;
    char name[NAME_LEN];
    drgb_t ambient;    /* Reflectivity for materials */
    drgb_t diffuse;
    double specular;
    double shininess;
    double transparency;
};