Computer Science 102
Lab 9

In this lab you will you will create C++ version of your linked list manager. You are *not required* to convert the list management components of your raytracer to C++ but you are free to do so if you wish. A sample *main.cpp, list.h* and sample input and output files *lab9.txt* and *lab9.log* are provided for you.

Your mission is to write a module named *list.cpp* which will contain the implementations of the following class methods.

```
class link_t
{
public:
   link_t(void);                  // default constructor
   link_t(void *);                // overloaded constructor
   ~link_t (void);                // destructor
   void    set_next(link_t *);    // used in adding new link
   link_t *get_next(void);        // retrieve the next pointer
   void   *get_entity(void);      // retrieve entity pointer

private:
   link_t *next;                  // next link in the list
   void   *entity;                // entity managed by link
};

class list_t
{
public:
   list_t(void);                   // constructor
   ~list_t (void);
   void   list_add(void *);        // add entity to end of list
   void   *list_start(void);       // set current to start of list
   void   *list_next(void);        // advance to next element in list

private:
   link_t *first;                 // first link
   link_t *last;                  // last link
   link_t *current;               // current link.
};
```

A discussion of the operation of the individual class methods follows and may be found in the class notes. You should leave the debugging code in the *link_t* destructor enabled.A discussion of the operation of each method follows.

link_t **methods**

This constructor is passed a pointer to the *obj_t* which this new link will own. It should set the *next* pointer to NULL and set the *entity* pointer to the entity being passed in:

```
link_t::link_t(void *newent)
{


}
```

The destructor should *free* the entity owned by the link. It *does not free the link_t.* That is done by the *delete* facility.

```
link_t::~link_t(void)
{
   fprintf(stderr, "Deleting %p\n", this);
}
```

The *set_next()* method is a typical "set" 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;*

```
void link_t::set_next(link_t *new_next)
{


}
```

link_t *getter* methods

The *get_next()* method is a typical “get” function that is used as a way to tell the *link_t* to return the value of own *next* pointer.

```
link_t * link_t::get_next()
{


}
```

The *get_object()* method is analogous. It would also work to simply make all of the *next* and *obj* 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. It should return the *entity* pointer.

```
void * link_t::get_entity()
{


}
```

list_t **class methods**

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

```
    list_t::list_t()
    {
       first = NULL;
       last =  NULL;
       current = first;
    }
```

The *list_t* destructor must *process the entire list* and *delete* each *link_t* instance. As in the C version, it is mandatory that the *address of the next link_t be remembered before the current one is deleted.*

```
list_t::~list_t(void)
{

}
```

Adding a new object to the list

The *list_add()* method creates a new *link_t* and passes its constructer a pointer to the entity to be attached to the *link_t*.   The *link_t* constructor returns a pointer to the new *link_t.*

If this is the *first* item added to an empty list, the *list_add* method should set the *first, last* and *current* pointers to the new link.

Otherwise, the *next* pointer of the existing *last* link should be set to point to the new *link_t* and the *last* pointer of the *list_t* should be set to the new *link_t.*

```
    void list_t::list_add(void *entity)
    {
       link_t *link;




    }
```

Retrieving the *first* element of the list

If the *list* is empty the *list_start()* method should return NULL. Otherwise, the *list_start* method sets the *current* pointer to the first *link* in the list and returns a pointer to the first *entity* in the list.

```
void * list_t::list_start(void)
{

}
```

Retrieving the next *obj_t* in the list.

The *next* method attempts to advance the *current* pointer. If the *current* pointer is already at the end of the list *NULL* will be returned. The use of the *persistent state variable current* will prove to be something of a pain in nested processing of the list. The function must return a pointer to the current *entity* in the list.

```
void_t * list_t::list_next(void)
{
   link_t *link;


}
```

Using the list_t class

Creating a new *list_t*

```
     list_t *list;
     list = new list_t();
```

Creating a new object and adding it to the list:

```
    sphere = new sphere_t(...);
    list->list_add((void *)sphere);
```

Deleting a *list* along with *all* the links and entities associated with the list:

```
     delete list;
```

Processing a list

The *start* method is used to set the internal *current* pointer to the internal *first* pointer and returns a pointer to the first object in the list.

The *next* method is used to advance *current* to point to the next *link_t* and return the *obj_t* pointed to by the new *link_t.* If *current* already points to the end of the list then *NULL* is returned.

```
   list_t *list = model->objs;
   object_t  *obj;
    :
   obj = (object_t *)list->start();
   while (obj != NULL)
   {
      obj->dumper(out);
      obj = (object_t *)list->next();
   }
```

This works well *unless* inside the loop there is a call to an inner function that also need to process the list. If the inner function uses the same *list_t* as the outer one, it will leave the value of *current* at *last* breaking the caller. We will return to this issue later.

In this lab you will submit a single file, list.cpp that includes the new class methods constructed as part of this lab.

```
     sendlab.102.labsection#  lab#  list.cpp
```