Linked Lists

Introduction

In 101, you saw how data is organized and processed sequentially using an array. You probably performed several operations on arrays, such as sorting, inserting, deleting, and searching. If data is not sorted, then searching for an item in the array can be very time-consuming, especially with large arrays. Once the data is sorted, we can use a binary search and improve the search algorithm; however, insertion and deletion become time-consuming, especially with large arrays, because these operations require data movement. With an array of fixed size, new items can be added only if there is room. Thus, there are limitations when organizing data in an array.

We can create powerful data structures by combining structures and pointers. One such structure is a linked list. A linked list is a collection of independent structures (often called nodes) that contain data. The individual nodes in the list are connected by links, or pointers. A program accesses the nodes in the list by following the pointers. Usually the nodes are dynamically allocated. A linked list is a very useful data structure and many of the techniques used to manipulate linked lists are applicable to other data structures.

We will use linked lists to organize material, objects, and lights in the raytracer.

The characteristics of the lists used by the raytracer include the following:

1. Newly created structures are always added to the end of the appropriate list
2. Individual structures are never deleted from the list
3. Lists are always processed sequentially from beginning to end
4. We desire a single generic mechanism that can manage lists of the three different structure types.
List data structures

The link_t structure

The typedef facility can be used to create an identifier for a user defined type. The following example creates a new type name, link_t, which is 100% equivalent to struct link_type

typedef struct link_type
{
  struct link_type *next; /* next link in the list */
  void *entity;    /* the entity(object_t, light_t) */
  /* that this link owns */
} link_t;

● There is a single instance of the link_t structure for each element in each list.
● Note we must use "official" name struct link_type when declaring next because link_t is not known to be a type definition until 3 lines later!
● As shown in the figure on the previous page each link contains two pointers;
  ● One is to the next link_t in the list.
  ● The other points to the actual entity being managed by the list
● The entity pointer is declared to be of type void *,
  ● A void * pointer is a pointer to something of unknown or generic type.
  ● In the raytracer, depending on the list being processed, the entity might be an object_t, a material_t or a light_t
  ● void * pointers can be freely assigned to other pointers
  ● void * pointers can never be directly used to access the memory to which they point because the size and type of the location is unknown.

Example:

material_t mat;
material_t *mloc = &mat;
void       *vloc = &mat;

Now vloc and mloc both point to the same instance of a material_t, but only mloc can be used to access the elements of the material_t structure.

mloc->ambient.r = 2.2;
vloc->ambient.r = 1.0;  <-- Won't work. Gives compile error
The `list_t` structure

typedef struct list_type
{
    link_t   *first; /* pointer to first link in list */
    link_t   *last;  /* pointer to last link in list */
    link_t   *current; /* current link in list */
}  list_t;

- There is a single instance of the `list_t` structure for each list.
- We will use three lists in the ray tracer: one for materials, one for visible objects, and one for lights.
Implementation

We will take an object oriented approach in building our list manager.

An object consists of a collection of:

● related data items, and
● functions or methods that can be used by external "users" to manipulate the data items;
● external "users" are not allowed to read or write the "private" data items associated with the object.

In C++ (or Java) an object is defined via a class definition.

● The link_t structure becomes a link_t class.
● It keeps the same data items (next and entity) but is augmented by methods that are used to manipulate them.
● The list_t structure would be defined in a separate class definition.

In C an object is represented by a structure that contains the related data items and a collection of functions that manipulate them. External users of the "object" should not directly reference the data items.

For now, our list management module (to be constructed in lab) will include the following functions:

The list_init() function used to create a new list. In a true O-O language, each class has a constructor method that is automatically invoked when a new instance of the class is created. The list_init() function serves this role here:

Its mission is to:

1 - malloc() a new list_t structure.
2 - set the first, and current and last elements of the structure to NULL.
3 - return a pointer to the list_t to the caller.

```c
list_t *list_init(void) {
}
```
The `list_add()` function must add the element pointed to by `new` to the list structure pointed to by `list`. Its mission is to:

1. `malloc()` a new instance of `link_t`,
2. add it to the end of the list,
3. ensure the `next` pointer of the new link is NULL and
4. ensure the `next` pointer of the `link_t` that used to be at the end of the list points to the new `link_t`
5. Set the current pointer to the new `link_t`

Two cases must be distinguished:

1. the list is empty (`list->first == NULL`)
2. the list is not empty (`list->first != NULL`)

```c
void list_add(list_t *list, void *new) {
}
```

The `list_reset()` function should set the current pointer to the first pointer. If the list is empty this will cause the current pointer to be set to NULL.

```c
int list_reset(list_t *list) {
}
```
The `list_not_end()` function should return 1 if the `current` pointer is not null and return 0 if the `current` pointer is NULL. Thus, `list_not_end()` should return(0) when either the list is empty or when the `current` pointer is advanced beyond the last link in the list.

```c
int list_not_end(list_t *list)
{
}
```

The `list_get_entity()` function should return the address of the entity pointed to by the `link` to which the `current` pointer points. The `list_not_end()` function should be called **BEFORE** `list_get_entity()` is invoked to make sure the current pointer points to a valid link! The call to `assert()` will abort the program if `list_get_entity()` is invoked in an improper state.

```c
void *list_get_entity(list_t *list)
{
    assert(list->current != NULL);
}
```

The `list_next_link()` function should advance the `current` pointer so that it points to the next link in the list. If the `current` pointer is presently pointing at the last link in the list, then this call will and should set the current pointer to NULL. The `list_next_link()` function should never be called when the `current` pointer is already NULL. Proper use of `list_not_end()` will ensure that this doesn't occur.

```c
void *list_next_link(list_t *list)
{
    assert(list->current != NULL);
}
```
Deleting a list

The `list_del()` function. This function should process the entire list. For each link in the list, it should

1 - invoke the `free()` function to free the `item` the link owns and then
2 - it should free the `link_t`.

Care must be taken *not to reference a `link_t` after it has been freed.* When all links and items are free, the `list` header itself should be freed.

```c
void list_del(list_t *list)
{
}
```
Processing a list

This code segment shows

- how to define an arbitrary structure that might be managed by the list
- how to ask `list_init()` to create a new list.
- how to process the list from first to last

```c
typedef struct entity_type
{
    char e_name[16];
    int  e_id;
} e_t;

eloc = list_init();
/* Load the list */
load_my_list(elist);

/* Now traverse the list printing attributes of the elements */
list_reset(elist);   // set current to first element
while (list_not_end(elist))
{
    eloc = (e_t *)list_get_entity(elist);
    printf("%s %d \n", eloc->e_name, eloc->e_id);
    list_next_link(elist);
}
```