Computer Science 102
Lab 10

In this lab you will you will extend the C++ version of your linked list manager. A sample *main.c, list.h and entity.h* are provided for you. You will note in *entity.h* that the *e_t* is now a true *C++* class.

Step 1: modify your *~link_t* destructor so that it uses *delete* instead of *free()* to delete the associated instances of the *e_t* class*.*

Step 2: make sure that your *list_add()* method sets the *current* pointer to the *last element* in the list (i.e. the one that was just added).

Step 3: implement the new methods *list_insert()* and *list_delete()* as class methods in the *list_t* class.

```
class list_t
{
public:
   list_t(void);                      // constructor
   ~list_t (void);
   void   list_add(void *);        // add entity to end of list
   void   list_insert(void *);     // insert entity before current
   void   list_delete(void );      // delete current entity
   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.
};
```
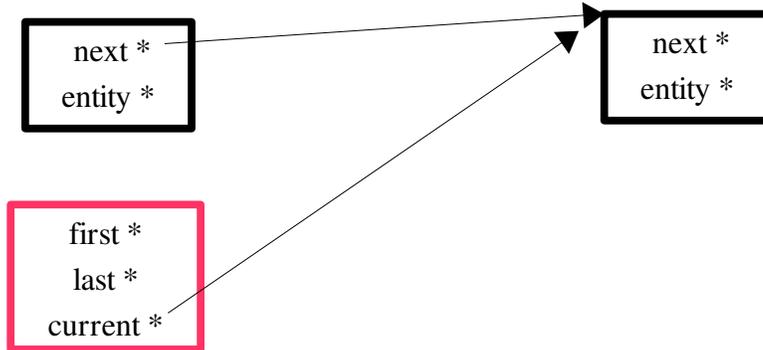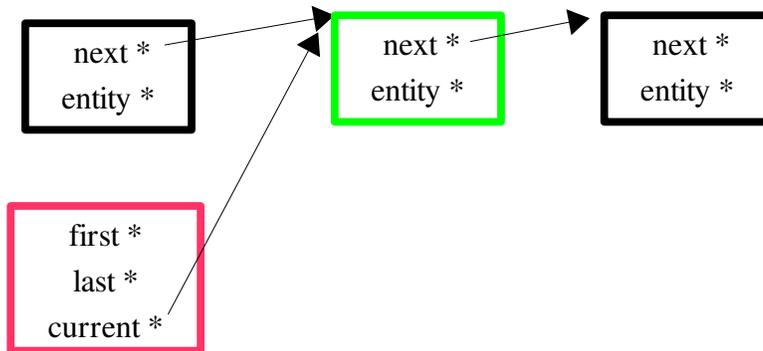
vlid list_insert(void *)

The *list_insert(void *)* method has a mission somewhat similar to *list_add()* in that does cause a new *e_t* to enter the list. However, unlike *list_add()* which always appends the new *link_t* to the end of the existing list*,* the *list_insert()* method must insert the new *link_t* directly in front of the existing *current* link.

When the *list_insert()* operation is complete the *next* pointer of the preexisting predecessor of *current* should point to the new *link_t,* the *next* pointer of the *new link_t* should have the previous value of *current* and *current* should have the address of the new *link_t.* In the diagram shown below existing *link_t's* are shown in black, the *list_t* in red and the inserted *link_t* in green.

BEFORE:

```
+-----------+                    +-----------+
| next *    |------------------->| next *    |
| entity *  |                    | entity *  |
+-----------+              ------>+-----------+
                         /
+-----------+          /
| first *   |        /
| last *    |      /
| current * |----
+-----------+
```

AFTER:

```
+-----------+      +-----------+      +-----------+
| next *    |----->| next *    |----->| next *    |
| entity *  |      | entity *  |      | entity *  |
+-----------+   -->+-----------+      +-----------+
             /
+-----------+
| first *   |
| last *    |
| current * |--
+-----------+
```
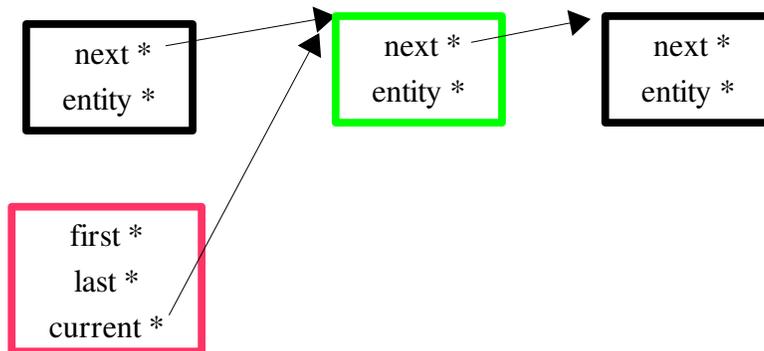
As with *list_add* the best way to make this work reliably is to have special case code that depends on the existing state of the list. For list_add() there were two relevant states *empty* and *not empty.* Here there are 3:

1 -   the list is presently empty (*current == first == last == NULL*). In this case *list_insert()* should just call *list_add()*

2 -   Current points to the start of the list (*current == first).* In this case the *next* pointer of the new *link_t* must point to the pre-existing first element and the *first* and *current* pointers set to the new *link_t.*

3 -   The current element is not first. In this case you will need a local variable *link_t *traverse.* Set *traverse* to *first* and then process the list until *traverse->getnext() == current.* At this point in the *BEFORE* picture *traverse* will point to the *link_t* on the *left* and *current* to the *link_t on the right.*
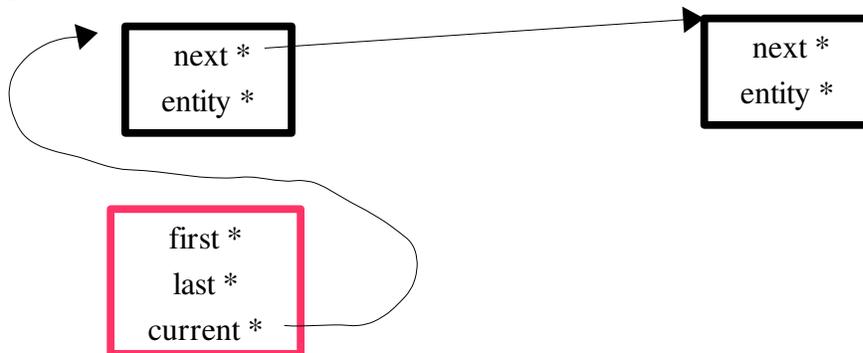
void list_delete(void)

The list_delete method should *delete* the *current* link_t. If the
deleted element is not the first element in the list, the *current*
pointer should point to the pre-existing predecessor of the deleted
link.   If the first link is deleted then *current* should point to its
successor.

BEFORE

```
┌──────────┐      ┌──────────┐      ┌──────────┐
│  next *  │─────▶│  next *  │─────▶│  next *  │
│          │◀─────│          │      │          │
│ entity * │      │ entity * │      │ entity * │
└──────────┘      └──────────┘      └──────────┘

      ┌──────────┐
      │ first *  │
      │          │
      │ last *   │
      │          │
      │ current *│
      └──────────┘
```

AFTER

```
          ┌──────────┐                    ┌──────────┐
    ─────▶│  next *  │───────────────────▶│  next *  │
    │     │          │                    │          │
    │     │ entity * │                    │ entity * │
    │     └──────────┘                    └──────────┘
    │
    │  ┌──────────┐
    │  │ first *  │
    │  │          │
    │  │ last *   │
    │  │          │
    └──│ current *│
       └──────────┘
```

Special cases to be considered include.

1-   List already empty (do nothing)
2-   List contains only a single element. Delete it and initialize
     first, last, and current
3-   Deleting the first element of a list that has more than one
     element (have to reset first)
4-   Deleting not the first element (need to traverse the list looking
     for the element that points to the current element)
4b- Deleting the last element of a list containing more than one
     element (have to reset last)

Debugging approach -

You will note that the *main.c* program has large sections disabled via the

#if 0
  disabled stuff

#endif

The proper way to debug this stuff is to enable the operations that are disabled *ONE AT A TIME*. Output produced with *all* code enabled may be found in lab10.txt.

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

    sendlab.102.labsection# lab# list.cpp