The doubly-linked list with iterators

Friday, November 19, 2010
9:03 AM

list.h:

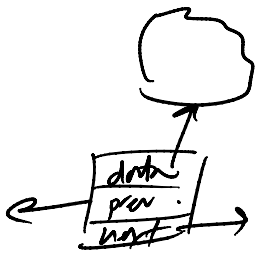
```
// forward declarations
template <typename T> class list_t;

template <typename T>
class list_t
{
    private:
    struct link_t
    {
        T data;
        link_t *prev;
        link_t * next;

        // constructor
        link_t (const T& d=T(), link_t *p = NULL, link_t *n = NULL):
                data(d), prev(p), next(n) { };
    };

    public:
    : TO BE FILLED IN (iterators, list_t constructor,
                       list_t destructor, operator=, iterator functions)

    // data members
    private:
    link_t *head;
    link_t *tail;
};
```



usage:

```
class data_t
{
    string name; ...
    int num;
};

list_t <data_t *> list;
```

T

calls constructor of data_t * a pointer

incoming arguments, so link_t constructor can be called link_t(d,p,n) or link_t()

Notice we have no first, last, current / / / marker marker

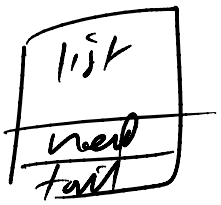
r
by record by title by iterator

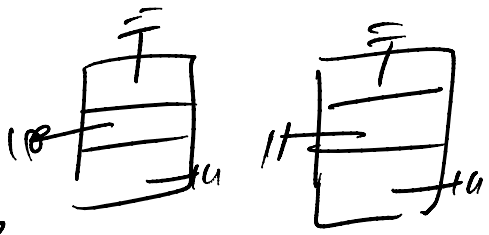
... still in class list_t definition

```
:
public:
 // constructors
list_t()
{
    head = new link_t;
    tail = new link_t;

    head->next = tail
    tail->prev = head;
}

~list_t()
{
    clear();
    delete head;
    delete tail;
}
```





malloc

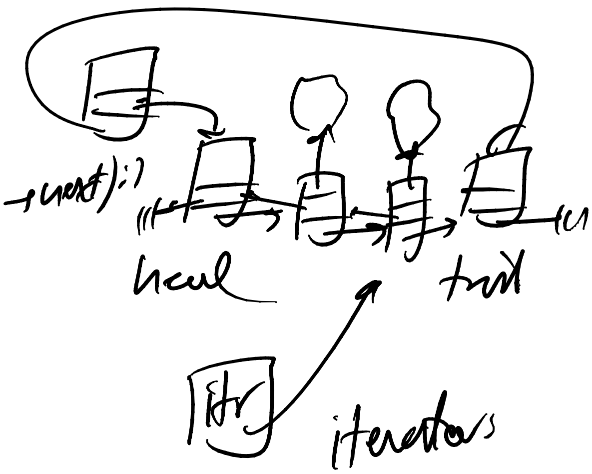
because we allocate memory, destructor ~list_t() must free this memory

write this member function to iterate thru link_t's, while list not empty, & frees each link_t

// iterator functions

iterator begin() { return iterator(head->next); }

iterator end()
{ return iterator(tail); }



iterators

fancy current ptr
but also an *object*
with its own constr,
destructor,
operator++
operator--

↑ constructor — ↑ data coming in to initialize iterator

In main.cpp:

for (itr = list.begin(); itr != list.end; itr++)

iterator erase(iterator);

iterator pop_front() { return erase(begin()); }

iterator insert(iterator, const T&);

— insert before iterator

Iterators:

in list.h

in class list_t { ... }

```
public:
class const_iterator
{
    public:
    // constructor with no arguments
    const_iterator() : curp(NULL) { },

    // public member functions *, ++, --, ==, !=
    const T& operator*() const   // accessor
    { return retrieve(); }

    const_iterator& operator++()   // prefix ++itr   (no args)
    { curp = curp->next; return *this; }

    const_iterator& operator++(int)   // postfix itr++
    { const_iterator old(curp); ++(*this); return old; }

    bool operator==(const const_iterator& rhs) const
    { return curp == rhs.curp; }

    // inline for !=
    // data member
    protected:
    list_t *curp;   // our old current ptr
```

copy constructor

```
T& retrieve() const
{ return curp->data; }
// constructor with link_t* argument
const_iterator(link_t *p) : curp(p) { };

friend class list_t<T>;
};
```

skeleton of const_iterator

```
class iterator : public const_iterator
{
  public:
  iterator()          { };  // let parent do the work

  T& operator*()                          // mutator
  { return iterator::retrieve(); }

  const T& operator*() const
  { return const_iterator::operator*(); }

  protected:
  iterator(link_t *p) : const_iterator(p) { }
};
```

Friday, November 19, 2010
9:56 AM

in list.cpp

include "data.h"
include "list.h"

template <typename T>
typename list_t<T>::iterator list_t<T>::erase(list_t<T>::iterator itr)
{
    list_t *p = itr.curr;
    typename list_t<T>::iterator ret(p->next);

    // erase item at itr
    p->prev->next = p->next;
    p->next->prev = p->prev;

    delete p;

    return ret;
}

// specialization
template class list_t<data *>;

↓ left out insert()