Basic doubly-linked list

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

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

private to list_t

generic type, can be *ptr

all-public class

node_t(const T& d=T(),

node_t *p = NULL,

node_t *n = NULL):

data(d), prev(p), next(n)

{};

}

default constr.

e.g. int()

usage: node_t()

or node_t(ptr, NULL, NULL)

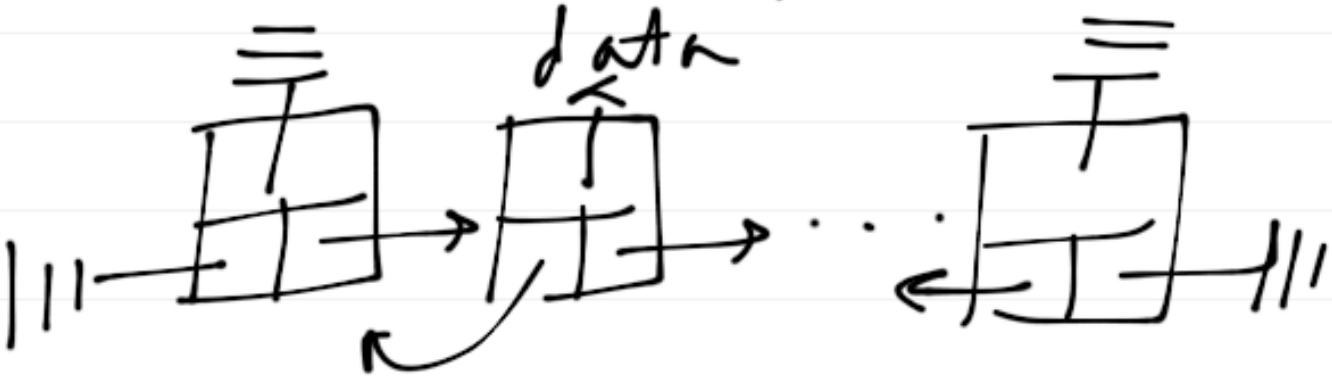
public:

node_t * begin()

{ return head->next; }

node_t * end()

{ return tail; }



head tail

(sentinel nodes)

see chp. 3

intended usage:

```
list_t   list;
node_t   *node;

for(node = list.begin();
    node != list.end();
    node = node->next;)
{
  // print *node
}
```

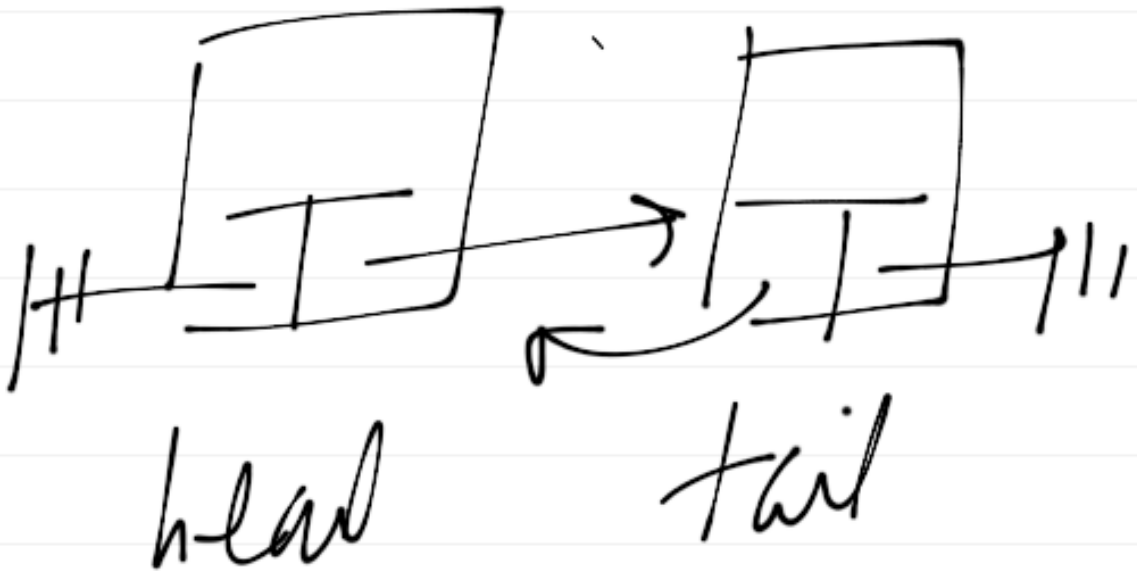
private

as private

violates info hiding

list_t list;

As expect list to be initialized to:



empty list: list.begin()
== list.end()

- node_t* implementation is C-style
- if we can't access node_t*, we can do:

```
list_t<int> list;
std::cout << "list: " << std::endl;
while (!list.empty()) {
    std::cout << list.front();
    list.pop_front()
}
```

What we want instead:

```
list_t<int>          list;
list_t<int>::iterator itr;
```

scope type instance

```
std::cout << "List" << std::endl;
for(itr = list.begin(); itr != list.end(); itr++)
    std::cout << *itr;
```

assignment op. operator != operator ++ operator *

list.begin() returns an iterator

↳ a fancy pointer, pointer wrapper

What if I have a fcn

print(const list_t<int> & list)

read-only access

⇒ const iterators

class list {

:

public

class const_iterator

{

// returns const ref

when *'ed (dereferenced)

:

public:

:

protected:

node_t * curr;

```
protected:
node_t * curr;

const_iterator (node_t *p):
    curr(p) { };
```

Constructor of const_iterator with node_t * argument (that will be inherited by derived class)

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

```
protected:
T& retrieve() const
{
    return curp->data;
}
```

---

back up in const_iterator;

```
public:
const T& operator*() const
{
    return retrieve();
}
```

public: //prefix ++itr

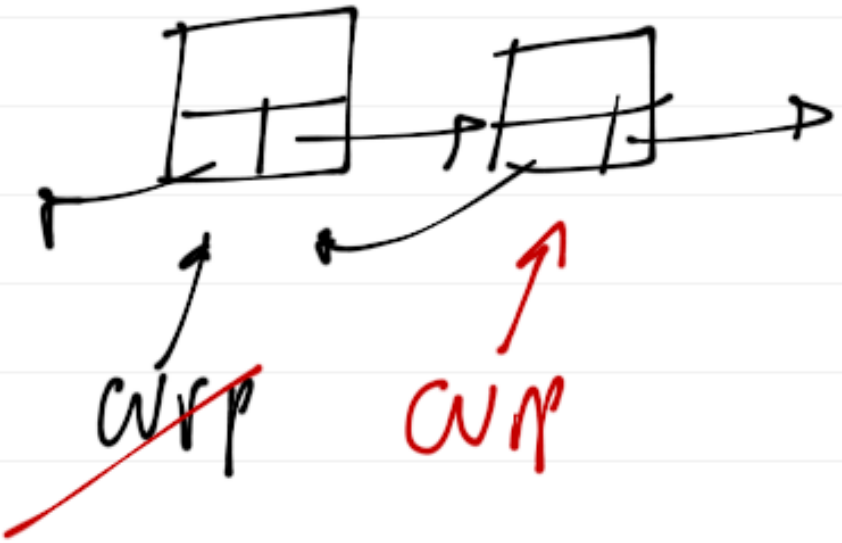
const_iterator& operator++()

{

curp = curp->next;

return *this;

}



// Postfix itr++

Const_iterator operator++(int)

{

Const_iterator old = *this;

++(*this);

return (old);

}

calling prefix operator++

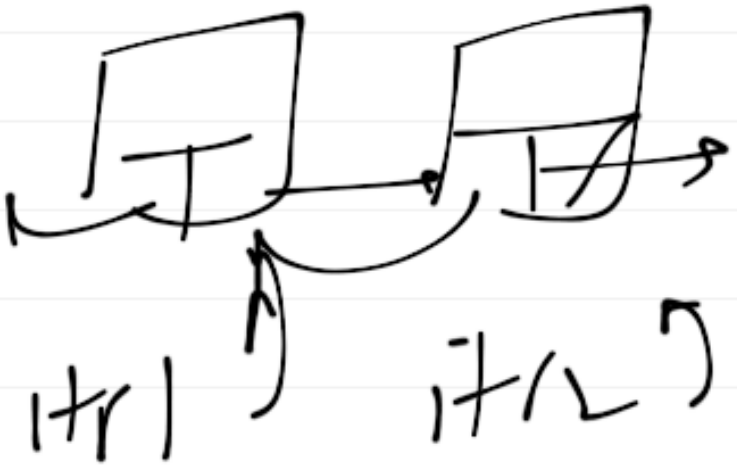
CVP

curr

old

return old

```
public:
const_iterator():
    curr(NULL) { };
bool operator==(const
    const_iterator& rhs)
                const
{ return curr == rhs.curr;
}
```



```
bool operator != (... rhs) const
{
    return !(curr == rhs.curr);
}
```

---

const_iterator : accessor

iterator : mutator

```
class iterator : public
        const_iterator
{
    public:
    iterator() { };

    T& operator*();

    {
        return iterator::
                retrieve();
    }
```

Scope: use my retrieve() fn, not parent's

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

Slope: Use parent's

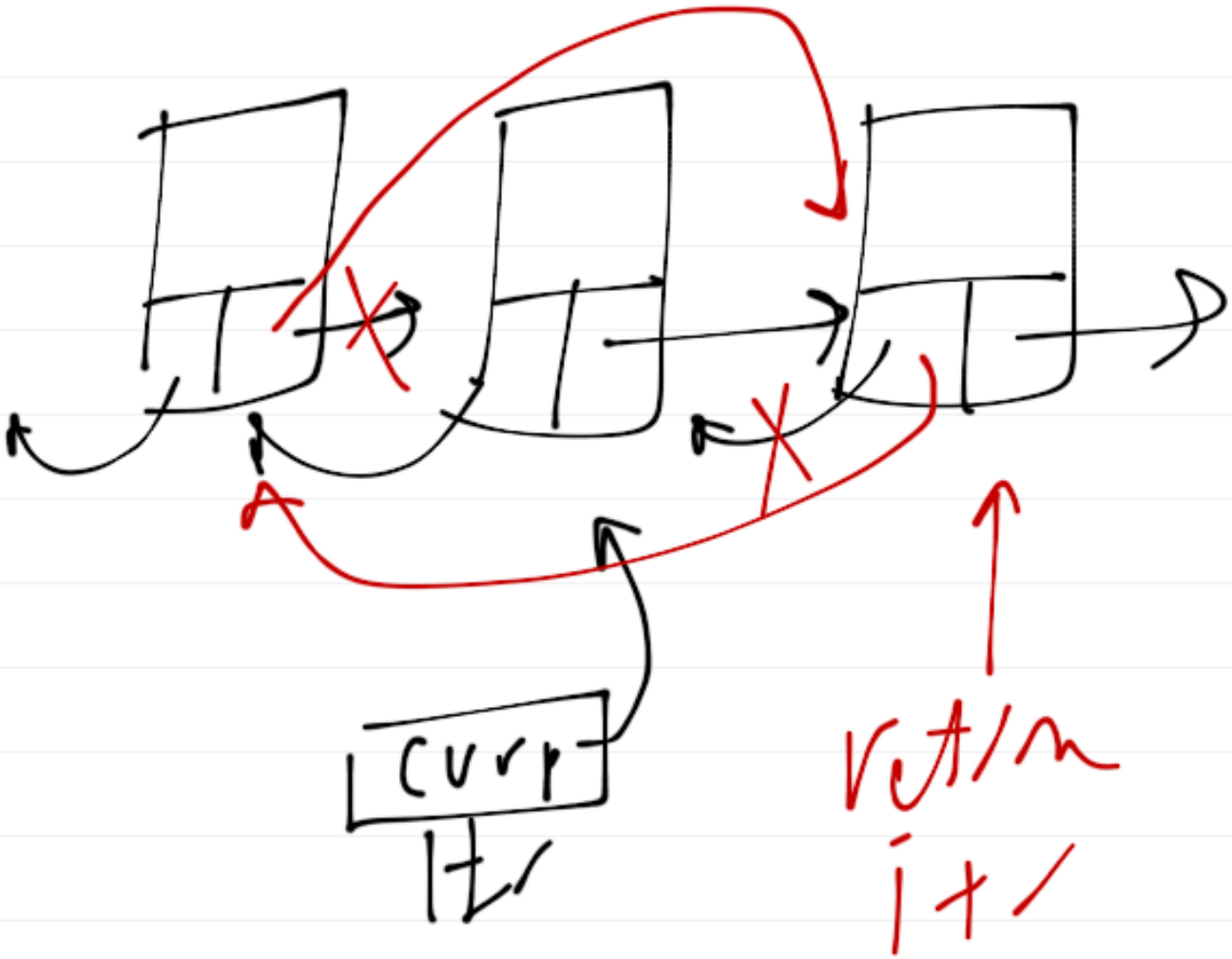
```
class list_t {
    :
  iterator begin()
    { return iterator(head->
                      next); }

  const_iterator begin()
    { return const_iterator(
                head->next);
    }
```

:

.

iterator erase(iterator);



in list.cpp:

```
template <typename T>
typename
list<T>::iterator

list_t<T>::erase(
    list_t<T>::iterator itr)
    typename list_t<T>::node_t
        *p = itr.curr;
```

```
typename list_t<T>::
    iterator ret(p->next);

  //erase
  p->prev->next = p->next;
  p->next->prev = p->prev;

  delete p;
  sz--;

  return ret;
}
```

Why is iterator
should give
count_iterator?

iterator (derived class)
expands or functionality
of base (parent) class
read + write over read