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

all-public class

generic type, can be a *ptr.

```
node_t(const T& d = T(),
```

default
const(.)
e.g. int()

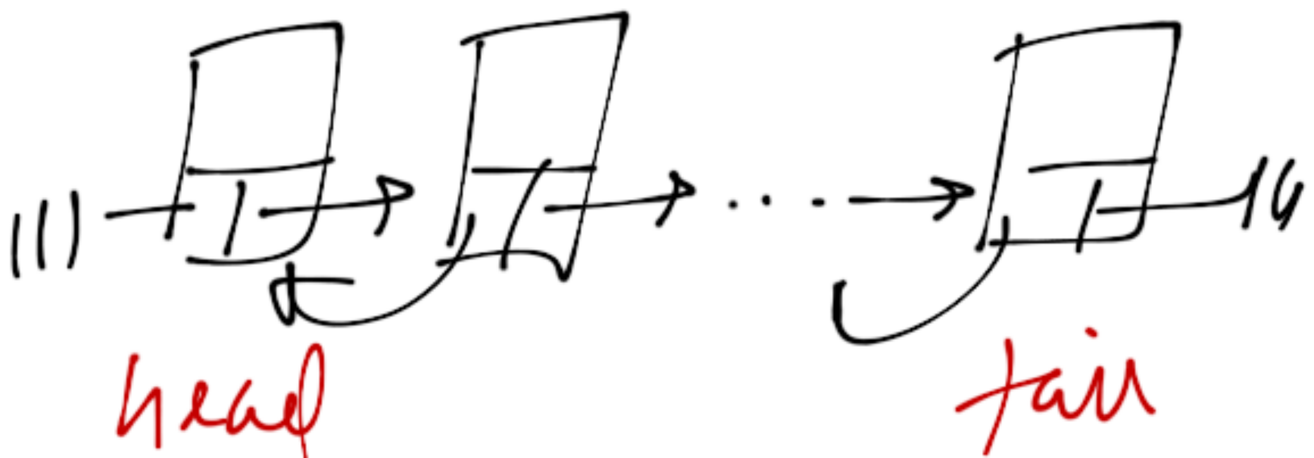
```
node_t *p = NULL,
node_t *n = NULL) :
    data(d), prev(p),
    next(n)
    { };
}
```

⋮

```
public:

node_t * begin()
    { return head->next; }

node_t * end()
    { return 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
}
```

it's private

also private

violates info hiding

so with our node_t &
implementation (C-style)

```
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 — `list_t<int>`

type — `iterator`

instance — `itr`

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

assignment op. — `=`

operator!= — `!=`

operator++ — `++`

operator* — `*`

list.begin() now returns
an iterator
↓
a fancy pointer,
or a pointer wrapper

---

what if I have a fcn
print(const list_t<int>&
list)

// can only access them
⇒ const iterators

```
class list_t {
        :
        :
    public:
    class const_iterator
    {   //returns const ref
        //   when *'ed
        //      (dereferenced)
        public:
          ...
        protected:
        node_t *curp;
```

```
protected:
node_t *curp;

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

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

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

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

---

back up in const_iterators

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

```
public:                              //prefix ++itr
const_iterator & operator++()
{  curr = curr->next;
   return *this;
}

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

calls prefix

```
public:
const_iterator() :
        curp(NULL) { };

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

const_iterator : accessor

iterator : mutator

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

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

scope: use ~~my~~ retrieve() fun

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

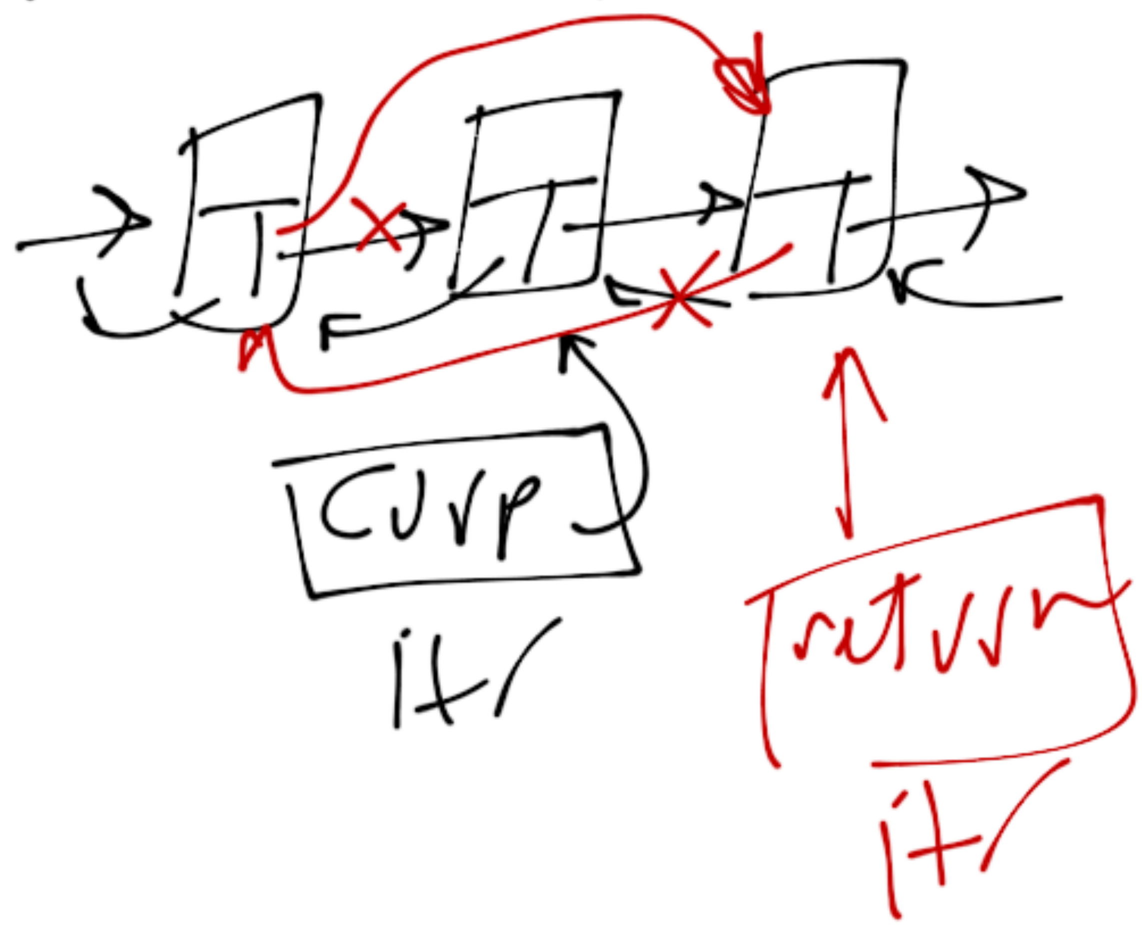
scope: use parent's

```
class list_t {
        ⋮

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

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

iterater erase(iterator);



in list.cpp:

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

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

  // erase

  p->prev->next = p->next;
  p->next->prev = p->prev;
  delete p;
  sz--;
  return ret;
}
```