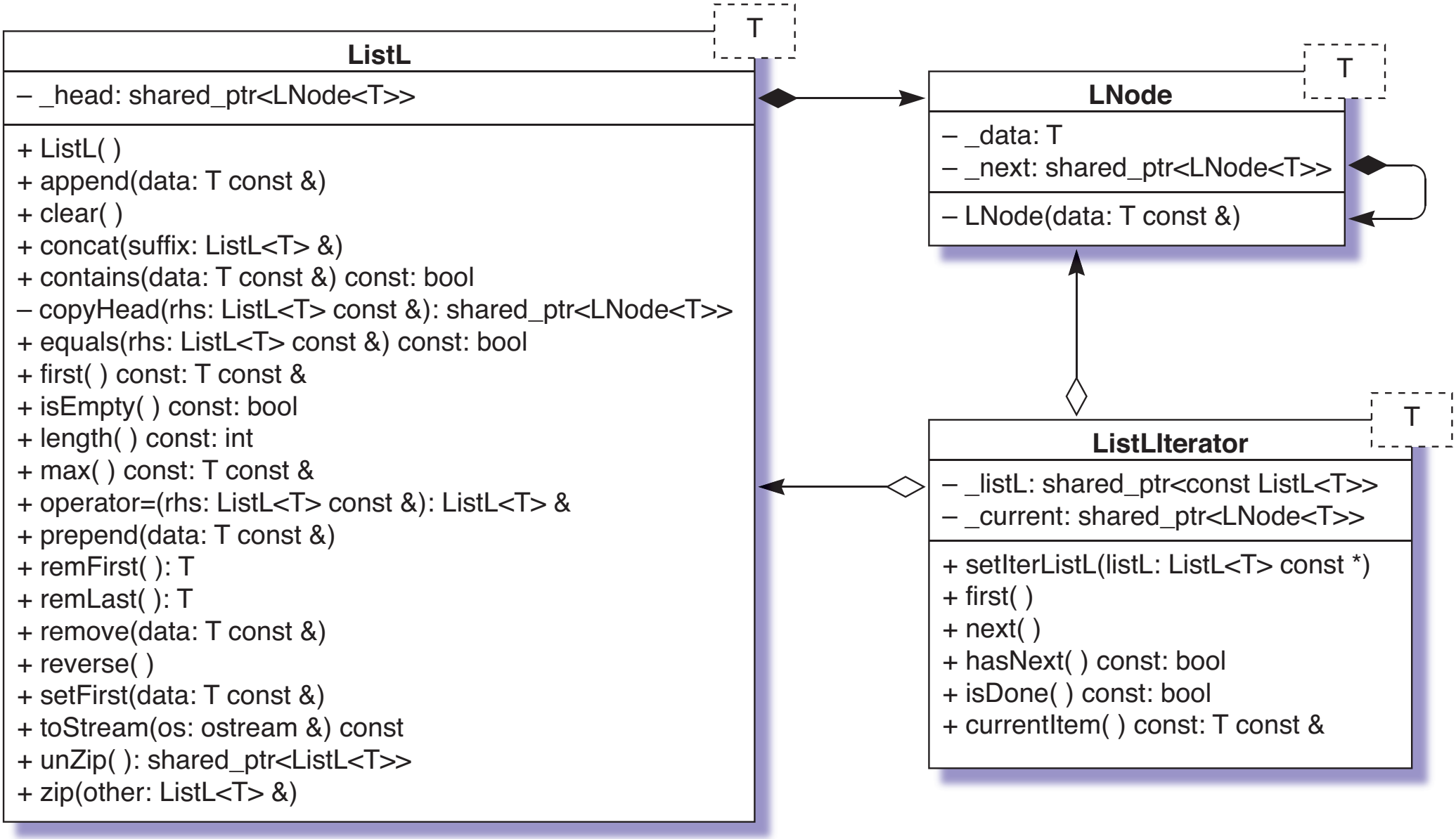
The Iterator Pattern

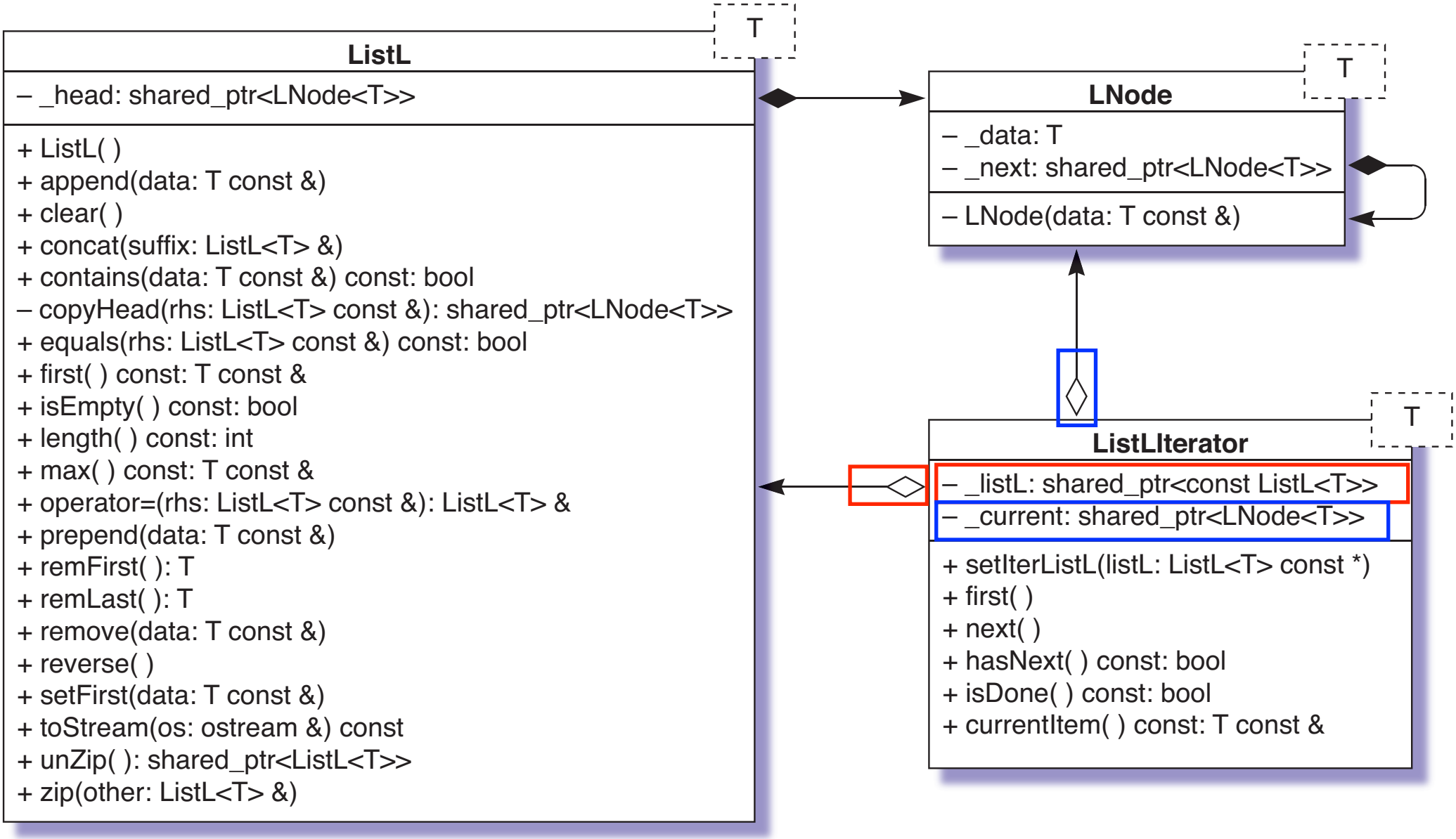
The iterator design pattern

Problem: The user wants to do some custom processing, e.g, find the sum or standard deviation.

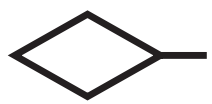
Cannot anticipate all the custom processing requests.

Solution: Provide the user with the ability to iterate through all the values of the list.


T
ListL
– _head: shared_ptr<LNode<T>>
+ ListL( )
+ append(data: T const &)
+ clear( )
+ concat(suffix: ListL<T> &)
+ contains(data: T const &) const: bool
– copyHead(rhs: ListL<T> const &): shared_ptr<LNode<T>>
+ equals(rhs: ListL<T> const &) const: bool
+ first( ) const: T const &
+ isEmpty( ) const: bool
+ length( ) const: int
+ max( ) const: T const &
+ operator=(rhs: ListL<T> const &): ListL<T> &
+ prepend(data: T const &)
+ remFirst( ): T
+ remLast( ): T
+ remove(data: T const &)
+ reverse( )
+ setFirst(data: T const &)
+ toStream(os: ostream &) const
+ unZip( ): shared_ptr<ListL<T>>
+ zip(other: ListL<T> &)
T
LNode
– _data: T
– _next: shared_ptr<LNode<T>>
– LNode(data: T const &)
T
ListLIterator
– _listL: shared_ptr<const ListL<T>>
– _current: shared_ptr<LNode<T>>
+ setIterListL(listL: ListL<T> const *)
+ first( )
+ next( )
+ hasNext( ) const: bool
+ isDone( ) const: bool
+ currentItem( ) const: T const &


T
ListL
– _head: shared_ptr<LNode<T>>
+ ListL( )
+ append(data: T const &)
+ clear( )
+ concat(suffix: ListL<T> &)
+ contains(data: T const &) const: bool
– copyHead(rhs: ListL<T> const &): shared_ptr<LNode<T>>
+ equals(rhs: ListL<T> const &) const: bool
+ first( ) const: T const &
+ isEmpty( ) const: bool
+ length( ) const: int
+ max( ) const: T const &
+ operator=(rhs: ListL<T> const &): ListL<T> &
+ prepend(data: T const &)
+ remFirst( ): T
+ remLast( ): T
+ remove(data: T const &)
+ reverse( )
+ setFirst(data: T const &)
+ toStream(os: ostream &) const
+ unZip( ): shared_ptr<ListL<T>>
+ zip(other: ListL<T> &)
T
LNode
– _data: T
– _next: shared_ptr<LNode<T>>
– LNode(data: T const &)
T
ListLIterator
– _listL: shared_ptr<const ListL<T>>
– _current: shared_ptr<LNode<T>>
+ setIterListL(listL: ListL<T> const *)
+ first( )
+ next( )
+ hasNext( ) const: bool
+ isDone( ) const: bool
+ currentItem( ) const: T const &

The aggregation relation

Symbol: 

Relation: Has-a-link-to-but-is-not-the-owner-of

Owner: Responsible for allocation and deallocation.

```
template<class T>
class ListLIterator {
private:
    shared_ptr<const ListL<T>> _listL;
    shared_ptr<LNode<T>> _current;
public:
    void setIterListL(shared_ptr<const ListL<T>> listL) {
        _listL = listL;
    }

    // Post: Positions the iterator to the first element.
    void first() { _current = _listL->_head; }

    // Post: Advances the current element.
    void next() { _current = _current->_next; }
```

```
    // Post: Checks whether there is a next element.
    bool hasNext() const { return bool(_current->_next); }

    // Post: Checks whether at end of the list.
    bool isDone() const { return !_current; }

    // Pre: The current element exists.
    // Post: The current element of this list is returned.
    T const &currentItem() const {
        if (!_current) {
            cerr << "currentItem precondition violated: "
                 << "Current element does not exist." << endl;
            throw -1;
        }
        return _current->_data;
    }
};
```

```
// ========== toStream ==========
template<class T>
void ListL<T>::toStream(ostream &os) const {
    os << "(";
    for (auto p = _head; p; p = p->_next) {
        if (p->_next) {
            os << p->_data << ", ";
        } else {
            os << p->_data;
        }
    }
    os << ")";
}
```

```
// ========= toStream4 =========
template<class T>
void ListL<T>::toStream4(ostream &os) const {
    ListLIterator<T> iter;
    iter.setIterListL(this->shared_from_this());
    os << "(";
    for (iter.first(); !iter.isDone(); iter.next()) {
        if (iter.hasNext()) {
            os << iter.currentItem() << ", ";
        } else {
            os << iter.currentItem();
        }
    }
    os << ")";
}
```