List

Introduction and idea

The never-ending debate: **Performance** or **Bug-proof** data structures, has been a thing since the very beginning of the programing. Nowadays there are some established practices for each one of the sides but sometimes that's not enough.

One of the most common data structures is list (abstract data type). In this project we will combine the 2 worlds into 1 and more specifically: a convenient and user-friendly structure List, supporting both the ideologies.

The main goal is simplicity for the user of our list structure – to have an easy way to "switch" between the ideologies. It is clear that the final product has to be bug-proof but we don't simply write end-product code, we have to go through the process of debugging. Therefore our structure supports 2 work mods – debug and performance. The idea is to use debug mod while in the process of debugging and once you have gone through all bugs and fixed them, simply by changing 1 line to have the performance mod on. The key idea is to use template so that the compilator makes the work for us - making the specifics required for each of the mods at **compile** time.

Debug Mode:

Simply supporting exceptions, though which, you can debug your program much easier than normally.

Performance Mode:

Does not make any unnecessary checking and focuses only on speed.

Definition and member types

```
template<
        typename T,
        bool debugMod = true
>List;
```

Template parameters:

T – The type of the elements

debugMod – indicator whether in debugMod (true) or not (false)

Member types:

value_type	T
size_type	Unsigned integer type (usually std::size_t)
difference_type	Signed integer type (usually std::ptrdiff_t)
reference	value_type&
const_reference	const value_type&
pointer	value_type*
const_pointer	const value_type*
iterator	LegacyBidirectionalIterator
const_iterator	Constant LegacyBidirectionalIterator
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

Member functions

The current release (2019.11.18) simply has most of the functionality std::list and more specifiacally:

```
List();
template<typename InputIt>
List(InputIt first, InputIt last);
List(const List& other);
List(List&& other);
~List();
List& operator=(const List& other);
List& operator=(List&& other);

reference front();
const_reference front()const;
reference back();
const_reference back()const;

iterator begin()noexcept;
const_iterator begin()const noexcept;
const_iterator cbegin()const noexcept;
iterator end()noexcept;
const_iterator end()const noexcept;
const_iterator cend()const noexcept;

iterator rbegin()noexcept;
const_iterator rbegin()const noexcept;
const_iterator crbegin()const noexcept;
iterator rend()noexcept;
const_iterator rend()const noexcept;
const_iterator crend()const noexcept;

bool empty()const noexcept;
size_type size() const noexcept;

void clear();

iterator insert(iterator pos, const T& value);
iterator insert(const_iterator pos, const T& value);
iterator insert(const_iterator pos, T&& value);
template<typename InputIt>
void insert(iterator pos, InputIt first, InputIt last);
template<typename InputIt>
iterator insert(const_iterator pos, InputIt first, InputIt last);
```

```
iterator erase(iterator pos);
iterator erase(const_iterator pos);
iterator erase(iterator first, iterator last);
iterator erase(const_iterator first, const_iterator last);

void push_back(const T& value);
void push_back(T&& value);
void pop_back();

void push_front(const T& value);
void push_front(T&& value);
void pop_front();

template <typename Compare = less<T> >
void merge(List& other, Compare comp = Compare());
template <typename Compare = less<T> >
void merge(List&& other, Compare comp = Compare());

void splice(const_iterator pos, List& other);
void splice(const_iterator pos, List&& other);
void splice(const_iterator pos, List& other, const_iterator it);
void splice(const_iterator pos, List&& other, const_iterator it);
void splice(const_iterator pos, List& other, const_iterator first, const_iterator last);
void splice(const_iterator pos, List&& other, const_iterator first, const_iterator last);

void remove(const T& value);
template<typename UnaryPredicate>
void remove_if(UnaryPredicate p);

void reverse() noexcept;

template<typename BinaryPredicate = equal_to<T> >
void unique(BinaryPredicate p = BinaryPredicate());

template<typename Compare = less<T> >
void sort(Compare comp = Compare());
```

Non-member functions

```
template<typename T, bool debugMod>
bool operator==( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
template<typename T, bool debugMod>
bool operator!=( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
template<typename T, bool debugMod>
bool operator<( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
template<typename T, bool debugMod>
bool operator<=( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
template<typename T, bool debugMod>
bool operator>( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
template<typename T, bool debugMod>
bool operator>=( const List<T, debugMod>& lhs, const List<T, debugMod>& rhs );
```