DLL Doubly Linked List

Denna laboration är en introduktion till länkade listor och iteratorer. Den fungerar i huvudsak på samma sätt som std:list i C++98[1]. men C++ standarden lämnar alltid så mycket som möjligt öppet för den som implementerar. Här har vi vissa krav på hur implementationen ska göras för att tydliggöra lämpliga sätt att implementera en lista. Bland annat så krävs:

- Minimal storlek på noderna (innehåller bara next och prev pekare samt data).
- Ingen onödiga allokeringar (innebär att "huvudet" ska var en medlemsvariabel i List).

För att förenkla så stryker vi många av funktionerna som std::list har, de väsentligaste strukna är:

- alla move operationer
- vi har bara forward iterator, stryker reverse.
- operator=, swap och splice behövs bara om ni vill ha VG

För att få listan att fungera med standardalgoritmerna i STL så behövs "typedefs" så att t.ex. en summeringsfunktion kan veta vilken typ av data listan innehåller.

Syftet med laborationen är att ge exempel på pekarhantering och visa att det finns containers som fungerar på andra sätt än std::string och std:vector.

Ni ska implementera en lista med huvud dvs. det ska finnas en klass ”Link<T>” som används som huvud i själva List klassen och som Node klassen ärver från från Link samt har en medlemsvariabel som håller det data (T) som ska lagras i listan.

A[2] container in STL consists of the container class itself and (several) iterator classes. The iterator classes has normally no visible name but it accessed through type definitions in the container, i.e. `List<C>::iterator`.
For simplicity, we in this document describe it as if the iterator is declared:

```
template<class T> class ListIter;
```

but you should probably implement it in another way.

Labben betygssätts med VG eller G. Förutom fler funktioner så krävs för VG att koden har hög läsbarhet, är kort och koncis, saknar onödiga minnesallokeringar och är snabb. Även för G så får man inte gör alltför ineffektiv kod eller helt onödiga minnesallokeringar.

Tänk på att skriva i C++ stil! T.ex. så finns nästan aldrig Get och Set funktioner i C++ klasser.

1.1.1 Cast och säkerhet i STL

Vi gör vår kod i enlighet med den praxis som finns i STL. T.ex. så kan man inkrementera en iterator så den är efter listan och sen avreferera den. I exemplet är List == std::list.

```
List<int> l; l.push_back(1); l.push_back(2);
List<int>::iterator it=l.begin(); ++it; ++it; std::cout<<*it;
```

Enligt c++ standarden så är det "undefined behavior" vilket betyder att vad som helst kan hände. Med VS i debug mode så blir det exception. Med VS i release mode så skrivs det ut 0 när jag testar. Koden i labben ska följa denna princip, resultatet blir att det behövs static_cast<Node*> på vissa platser. Om vi skulle avvika från STL stilen så skulle vi lägga till en virtuell destructor i Link och sedan i stället göra dynamic_cast<Node*>. Kostnaden för detta är dels någon extra runtime för att kolla att det är en Node men värre är att det behövs en extra pekare i varje Link.

[1] C++11 list har andra krav på komplexiteten, med C++98 blir implementeringen mer generell.
[2] Av historiska skäl så är det en blandning av svenska och engelska i beskrivningen.

2 List klassen

List class members. See std::list for the exact definition of the functions.
[const] anger att funktionen finns i två versioner, med reso. utan const.
Grönmarkerade dunktioner krävs bara för VG

Works as std:list if nothing else is noted	Comments
`using iterator = ... , const_iterator = ...`	typdefs
`~List();`	
`List();`	
`List(const List& other);`	
`List(const char* other);`	For testing, should generate one node for each character.
`operator=(const List& other)`	Implementeras med copy constructor och swap
`[const] T& front()[const];`	
`[const] T& back() [const];`	
`iterator begin() noexcept;`	
`iterator end() noexcept;`	
`bool empty() const noexcept;`	
`size_t size() const noexcept;`	Obs: O(n) timekomplexity
`iterator insert(iterator pos, const T& value);`	insert a new node before pos.
`iterator erase( const iterator& pos );`	Delete the node pos points to from the list
`void push_back(const T& value);`	insert last
`void push_front(const T& value);`	insert first
`void pop_back();` `void pop_front();`	Delete last (first) node.
`void swap(List<T>& rhs );`	friend swap anropar denna swap. Obs O(1)
`void splice( const_iterator pos, list& other, const_iterator first, const_iterator last);`	se std::list för förklaring `other` används inte
`friend bool operator==(const List& lhs, const List& other)`	global function
`friend bool operator<(const List& lhs, const List& other)`	global function
och alla de andra (!=,>,<=,>=). Observera att om du implementerar == och < så kan de andra uttryckas med dem, t.ex. är x<=y det samma som !(y<x)	globala functions <=> Kanske vi ska använda i stället!
`friend std::ostream& operator<<(std::ostream& cout , const List& other);`	For testing
`#define CHECK assert(Invariant());`	Macro used inside the class.
`bool Invariant();`	public function for testing use.
`friend void swap(List<T>& lhs, List<T>& rhs );`	global function. Obs: O(1)

3 Iterator klassen (Bidirectional iterator)

See https://en.cppreference.com/w/cpp/named_req/BidirectionalIterator for details.
Described here are the “iterator”, the simplest way to make a const_iterator is to have iterotr take a template parameter X that is T for a iterator and const T for a const_iterator. ListIter is used as name of the class, you should probably use another class name.

Functions/typedefs	Comments
`iterator_category, value_type, difference_type, reference, pointer`	typdefs needed to make the standard algorithms work.

Functions/typedefs	Comments
`ListIter(Node<T>* p);`	You should change Node to whatever is needed in your program
`ListIter();`	
`ListIter(const ListIter& other);`	
`ListIter& operator=(const ListIter& other);`	
`T & operator*();`	
`T* operator->();`	
`ListIter& operator++();`	`++it;`
`ListIter& operator--()`	`--it`
`ListIter operator++(int);`	`it++`
`ListIter operator--(int);`	`it--`
`friend bool operator==(const ListIter& lhs, const ListIter& rhs);`	global function
`friend bool operator!=(const ListIter& lhs, const ListIter& rhs);`	global function

4 Tips, en tänkbar struktur är:

```
template<class T>
class List {
    class Node;

    class Link {
        friend class List<T>;
        Link* _next, *_prev;
        Link() :_next(this), _prev(this) {}
```

Lägg in funktioner för Insert och Erase här, enklast att hantera i Link samt splice för VG

```
    };

    class Node : public Link {
        friend class List;
        T _data;
    public:
        Node(const T& data) :_data(data) {};
    };

    template<class X> //X is T or const T
    class ListIter {
        friend class List;
        Link* _ptr;
    public:
```

Och här kommer alla funktioner i ListIter

```
    };

    Link _head;
```

Och här kommer alla funktioner i List

```
};
```

Tanken med att lägga in koden för Insert och Erase i link klassen är att man då kan göra både push_back, push_front och insert som anrop på Links Insert.
Prev och Next är hjälp funktioner som kan förenkla resten av programmet. Observera att de kan ge en pekare till själva huvudet (_head) som resultat.