STL

Тема 3. Контейнеры.

Часть 2. Последовательные контейнеры. List.

List

- Отсутствует итератор произвольного доступа
- Константное время вставки и удаления
- Склейка за константное время

template < class Type**, class** Allocator=**allocator<**Type**> > class list**

List. Typedefs

Typedef	Описание
const_iterator	Константный итератор
const_pointer	*const Type*.* В общем случае определен аллокатором
const_reference	*const Type&.* В общем случае определен аллокатором
const_reverse_iterator	Константный обратный итератор
difference_type	Знаковый целочисленный тип, который может определеть разность между двумя итераторами
iterator	итератор
pointer	*Type*.* В общем случае определен аллокатором
reference	*Type&.* В общем случае определен аллокатором
reverse_iterator	Обратный итератор
size_type	Тип представляющий количество элеменотов
value_type	*Type.*

List. Constructor

```
list( );
```

```
explicit list( size_type _Count );
```

```
list( size_type _Count, const Type& _Val );
```

```
list( const list<Type, Allocator>& _Right );
```

```
template<class InputIterator>
list( InputIterator _First, InputIterator _Last );
```

List. Constructor

```
list c0 list <int> c0;  // Create an empty

list <int> c1( 3 );  // Create a list c1 with 3 elements of default value 0

list <int> c2( 5, 2 );  // Create a list c2 with 5 elements of value 2

list <int> c4(c2);   // Create a copy, list c4, of list c2

// Create a list c5 by copying the range c4[_First, _Last)
c4_Iter = c4.begin( );
c4_Iter++;
c4_Iter++;
list <int> c5( c4.begin( ), c4_Iter );
```

List:: push_back\push_front\ pop_back\pop_front

```
void push_back( const Type& _Val );
void push_front( const Type& _Val );
void pop_back( );
void pop_front( );
```

```
list <int> c1;

c1.push_back( 1 );
c1.push_front( 2 );

c1.pop_back( );
c1.pop_front( );
```

List::insert

iterator insert(iterator _Where**, const Type&** _Val **);**

void insert(iterator _Where**, size_type** _Count**, const Type&** _Val **);**

template<class InputIterator>
void insert(iterator _Where**, InputIterator** _First**, InputIterator** _Last **);**

```
list <int> c1, c2;
c1.push_back( 10 ); c1.push_back( 20 ); c1.push_back( 30 );
c2.push_back( 40 ); c2.push_back( 50 ); c2.push_back( 60 );

list <int>::iterator Iter = c1.begin( );
Iter++;
c1.insert( Iter, 100 );

c1.insert( Iter, 2, 200 );

c1.insert( ++c1.begin( ), c2.begin( ),--c2.end( ) );
```

List::erase

iterator erase(iterator _Where **);**

iterator erase(iterator _First**, iterator** _Last **);**

```
list <int> c1;

c1.push_back( 10 ); c1.push_back( 20 ); c1.push_back( 30 );
c1.push_back( 40 ); c1.push_back( 50 );

c1.erase( c1.begin( ) );

list <int>::iterator Iter = c1.begin( );
Iter++;
c1.erase( Iter, c1.end( ) );
```

List::remove

void remove(const Type& _Val);

```
list <int> c1;

c1.push_back( 5 );
c1.push_back( 100 );
c1.push_back( 5 );
c1.push_back( 200 );
c1.push_back( 5 );
c1.push_back( 300 );

list <int> c2 = c1;
 c2.remove( 5 );
```

List::splice

```
void splice( iterator _Where, list<Type, Allocator>& _Right );
void splice( iterator _Where, list<Type, Allocator>& _Right, iterator _First );
void splice( iterator _Where, list<Type, Allocator>& _Right,
                              iterator _First, iterator _Last );
```

```
list <int> c1, c2, c3, c4;
c1.push_back( 10 ); c1.push_back( 11 );
c2.push_back( 12 ); c2.push_back( 20 ); c2.push_back( 21 );
c3.push_back( 30 ); c3.push_back( 31 );
c4.push_back( 40 ); c4.push_back( 41 ); c4.push_back( 42 );

list <int>::iterator w_Iter = c2.begin( ); w_Iter++;
c2.splice( w_Iter,c1 );

c2.splice( w_Iter,c3, c3.begin( ));

list <int>::iterator f_Iter = c4.begin( ); list <int>::iterator l_Iter = c4.end( ); l_Iter--;
 c2.splice( w_Iter,c4, f_Iter, l_Iter );
```

List::sort

```
void sort( );

template<class Traits>
void sort( Traits _Comp );
```

```
list <int> c1;

c1.push_back( 20 ); c1.push_back( 10 ); c1.push_back( 30 );

c1.sort( );

c1.sort( greater<int>( ) );
```

List::merge

```
void merge( list<Type, Allocator>& _Right );


template<class Traits>
void merge( list<Type, Allocator>& _Right, Traits _Comp );
```

```
list <int> c1, c2, c3;

c1.push_back( 3 ); c1.push_back( 6 );
c2.push_back( 2 ); c2.push_back( 4 );
c3.push_back( 5 ); c3.push_back( 1 );

c2.merge( c1 );

c2.sort( greater<int>( ) );
c2.merge( c3, greater<int>( ) );
```

List::unique

```
void unique( );


template<class BinaryPredicate>
void unique( BinaryPredicate _Pred );
```

```
list <int> c1;
not_equal_to<int> mypred;

c1.push_back( -10 ); c1.push_back( 10 ); c1.push_back( 10 );
c1.push_back( 20 ); c1.push_back( 20 ); c1.push_back( -10 );

list <int> c2 = c1; c2.unique( );

list <int> c3 = c2; c3.unique( mypred );
```

Аксессоры

- **iterator begin();**
- **iterator end( );**
- **iterator rbegin();**
- **iterator rend( );**
- **size_type size( ) const;**
- **size_type max_size( ) const;**
- **bool empty( ) const;**
- **reference front( );**
- **reference back( );**

List::assign

```
template<class InputIterator>
void assign( InputIterator _First, InputIterator _Last );

void assign( size_type _Count, const Type& _Val );
```

```
list<int> c1, c2;
c1.push_back( 10 ); c1.push_back( 20 ); c1.push_back( 30 );
c2.push_back( 40 ); c2.push_back( 50 ); c2.push_back( 60 );

c1.assign( ++c2.begin( ), c2.end( ) );

c1.assign( 7, 4 );
```

List::swap

```
void swap( list<Type, Allocator>& _Right );

friend void swap( list<Type, Allocator>& _Left,
                  list<Type, Allocator>& _Right )
```

```
list <int> c1, c2, c3;

c1.push_back( 1 ); c1.push_back( 2 ); c1.push_back( 3 );
c2.push_back( 10 ); c2.push_back( 20 );
c3.push_back( 100 );

c1.swap( c2 );

swap( c1,c3 );
```

Практическое задание

Реализовать алгоритм сортировки вставками для

1. Целых чисел по убыванию
2. Списка векторов по длине