Язык C++

STL. Итераторы и основные алгоритмы - III

Адаптеры контейнеров

- template<class T, class Container = std::deque<T>> class stack;
- template<class T, class Container = std::deque<T>> class queue;
- template<class T, class Container = std::vector<T>,
  class Compare = std::less<typenameContainer::value_type>>
  class priority_queue;

std::stack

```
template<typename T, typename Container=std::vector<T>>
class CMyStack {
public:
   typedef typename Container::value_type      value_type;
   typedef typename Container::reference       reference;
   typedef typename Container::const_reference const_reference;
   typedef typename Container::size_type       size_type;

   void push(const value_type& value) { data_.push_back(value); }

   void pop() { data_.pop_back();}

   bool empty() const { return data_.empty(); }

   const_reference top() const { return data_.back();}

private:
   Container data_;
};
```

Адаптеры итераторов

- back_insert_iterator<Container> (push_back)
- front_insert_iterator<Container> (push_front)
- insert_iterator<Container> (insert)

```
int main() {
   int arr[] = {1,2,3,4,5};
   std::vector<int> v;

   std::copy(arr, arr + 5, std::back_inserter(v));

   return 0;
}
```

back_insert_iterator

// Реализовываем (LegacyOutputIterator)

//

Потоковые итераторы

- istream_iterator
  - Ввод
  - Входной, но не выходной итератор
- ostream_iterator
  - Вывод
  - Выходной, но не входной итератор

Потоковые итераторы

```
int main() {
   std::vector<int> v ;

   std::copy(
       std::istream_iterator<int>(std::cin),
       std::istream_iterator<int>(),
       std::back_inserter<std::vector<int>>(v)
   );

   std::copy(v.begin(),  v.end(),  std::ostream_iterator<int>(std::cout, " "));

   return 0;
}
```

Tag Dispatch Idiom

```
struct tag_1 {};
struct tag_2 {};
struct tag_3 : public tag_2 {};

struct TypeA {};
struct TypeB {};
struct TypeC {};

template<typename T>
struct my_traits {
   typedef tag_1 tag;
};

template<>
struct my_traits<TypeB> {
   typedef tag_2 tag;
};

template<>
struct my_traits<TypeC> {
   typedef tag_3 tag;
};
```

Tag Dispatch Idiom

```
template<typename T>
void func_dispatch(const T& value, const tag_1&) {
   std::cout << "tag1\n" ;
}

template<typename T>
void func_dispatch(const T& value, const tag_2&) {
   std::cout << "tag2\n" ;
}

template<typename T>
void evaluate(const T& value) {
   func_dispatch(value, typename my_traits<T>::tag());
}
```

iterator_traits

```
int main () {
   std::vector<int> v = {1,2,3,4,5};

   std::iterator_traits<std::vector<int>::iterator> tr;

   auto it = std::find(v.begin(), v.end(), 3);

   /*
   template<typename _Iterator, typename _Predicate>
   inline _Iterator
   __find_if(_Iterator __first, _Iterator __last, _Predicate __pred)
   {
     return __find_if(__first, __last, __pred,
              std::__iterator_category(__first));
   }
   */

   return 0;
}
```

input_iterator_tag

```
struct input_iterator_tag { };

struct output_iterator_tag { };

struct forward_iterator_tag : public input_iterator_tag { };

struct bidirectional_iterator_tag : public forward_iterator_tag { };

struct random_access_iterator_tag : public bidirectional_iterator_tag { };

struct contiguous_iterator_tag: public random_access_iterator_tag { };
```

Iterator Operation

- advance
- distance
- next
- prev

Iterator operation

```
template<class It>
typename std::iterator_traits<It>::difference_type
   distance(It first, It last)
{
   return detail::do_distance(
            first, last,
            typename std::iterator_traits<It>::iterator_category()
          );
}
```

Iterator operation

```
namespace detail {
   template<typename It>
   typename std::iterator_traits<It>::difference_type
   do_distance(It first, It last, std::input_iterator_tag) {
       typename std::iterator_traits<It>::difference_type result = 0
       while (first != last) {
           ++first;
           ++result;
       }
       return result;
   }

   template<class It>
   typename std::iterator_traits<It>::difference_type
   do_distance(It first, It last, std::random_access_iterator_tag) {
       return last - first;
   }
}
```