Итераторные адаптеры

Итераторные адаптеры

Итераторные адаптеры - итераторы-обертки над стандартными итераторами и контейнерами с особой семантикой операций.

Проблема. Допустим, хотим скопировать элементы одного контейнера *в конец* другого:

```
std::vector<int> v{1, 2, 3};
std::list<int> l{4, 5, 6};

std::copy(l.begin(), l.end(), v.end());  // <-- что пойдет не так?
```

```
template <class SrcIt, class DstIt>
DstIt copy(SrcIt begin, SrcIt end, DstIt dest) {
  for (; begin != end; ++begin, ++dest) {
    *dest = *begin;
  }
  return dest;
}
```

Итераторные адаптеры

Проблема. Допустим, хотим скопировать элементы одного контейнера *в конец* другого:

```
std::vector<int> v{1, 2, 3};
std::list<int> l{4, 5, 6};

std::copy(l.begin(), l.end(), v.end());  // <-- что пойдет не так?
```

```
template <class SrcIt, class DstIt>
DstIt copy(SrcIt begin, SrcIt end, DstIt dest) {
  for (; begin != end; ++begin, ++dest) {
    *dest = *begin;
  }
  return dest;
}
```

Итератор `v.end()` нельзя разыменовывать и инкрементировать! Элементы не добавляются, а заполняют ячейки не принадлежащие контейнеру.

`std::back_insert_iterator`

Хотелось бы, чтобы присваивание результату разыменования итератора вызывало `push_back`.

Для этого есть специальный адаптер `std::back_insert_iterator`.

```
std::vector<int> v{1, 2, 3};
std::list<int> l{4, 5, 6};
std::back_insert_iterator<std::vector<int>> back_it(v);

std::copy(l.begin(), l.end(), back_it);  // <-- вставка в конец
```

```
template <class SrcIt, class DstIt>
DstIt copy(SrcIt begin, SrcIt end, DstIt dest) {
  for (; begin != end; ++begin, ++dest) {
    *dest = *begin;  // <=> v.push_back(*begin);
  }
  return dest;
}
```

`std::back_insert_iterator`: `std::back_inserter`

Для упрощения создания объекта можно воспользоваться функцией `std::back_inserter`.

```
std::vector<int> v{1, 2, 3};
std::list<int> l{4, 5, 6};

std::copy(l.begin(), l.end(), std::back_inserter(v));  // <-- вставка в конец
```

```
template <class SrcIt, class DstIt>
DstIt copy(SrcIt begin, SrcIt end, DstIt dest) {
    for (; begin != end; ++begin, ++dest) {
        *dest = *begin;  // <=> v.push_back(*begin);
    }
    return dest;
}
```

`std::back_insert_iterator`: как работает

```
template <class Container>
class back_insert_iterator {
  Container* container_;

 public:
  explicit back_insert_iterator(Container& container) : container_(&container) {}
  back_insert_iterator& operator=(const typename Container::value_type& value) {
    container->push_back(value);
    return *this;
  }
  back_insert_iterator& operator=(typename Container::value_type&& value) {
    container->push_back(std::move(value));
    return *this;
  }
  back_insert_iterator& operator*() { return *this; }
  back_insert_iterator& operator++() { return *this; }
  back_insert_iterator operator++(int) { return *this; }
};

template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& cont) {
  return back_insert_iterator<Cont>(cont);
}
```

Адаптеры

Помимо `std::back_insert_iterator`, есть `std::front_insert_iterator` и `std::insert_iterator` (с порождающими функциями `std::front_inserter` и `std::inserter`).

Последний позволяет осуществлять вставку в произвольное место контейнера:

```
std::vector<int> v{1, 2, 3};
std::list<int> l{4, 5, 6};

std::copy(l.begin(), l.end(), std::inserter(v, std::next(v.begin(), 2)));
// v = [1, 2, 4, 5, 6, 3]
```

Обратный проход

А что, если хочется обойти контейнер в обратном порядке?

```
for (auto it = c.end(); rend = c.begin(); it != rend;) {
  --it;
  std::cout << *it << ' ';
}
```

Для этого у каждого контейнера, который поддерживает двунаправленный итератор, есть специальный тип `reverse_iterator`, который подменяет операции `++`, `--`, `+`, `-` операциями `--`, `++`, `-`, `+` соответственно:

```
for (auto it = c.rbegin(); it != c.rend(); ++it) {
  std::cout << *it << '\n';
}
```

Сортировка в обратном порядке:

```
std::sort(v.rbegin(), v.rend());
```

`reverse_iterator` реализация

Как реализовать reverse_iterator для своего контейнера?

Очень просто - необходимо написать класс, в котором будет храниться обычный итератор

```
class reverse_iterator {
  my_iterator it_;
  // ...
};
```

`reverse_iterator` реализация

Разыменование оставить без изменений:

```
class reverse_iterator {
  // ...
  value_type& operator*() {
    return *it_;
  }

  value_type* operator->() {
    return it_.operator->();
  }
  // ...
};
```

`reverse_iterator` реализация

Затем переопределить инкремент и декремент:

```
class reverse_iterator {
  // ...
  reverse_iterator& operator++() {
    --it_;
    return *this;
  }

  reverse_iterator& operator--() {
    ++it_;
    return *this;
  }
  // ...
};
```

`reverse_iterator` реализация

Конечно, никто этой ерундой заниматься не собирается

`std::reverse_iterator`

Для этого есть соответствующий адаптер - `std::reverse_iterator`.

Он принимает в качестве шаблонного параметра тип обычного итератора и переопределяет методы нужным образом:

```
std::vector<int>::iterator it = /* ... */;
std::reverse_iterator<std::vector<int>::iterator> rit(it);

++rit;      // <=> --it
--rit;      // <=> ++it
rit += 5;   // <=> it -= 5
```

`std::reverse_iterator`

Важно понимать, что элемент, на который указывает `reverse_iterator` фактически отличается от того, на который указывает обычный итератор:

Обычный итератор `it`:

```
1 2 3 -> 4 5  // указывает на 4
```

Обратный итератор `rit(it)`:

```
1 2 3 <- 4 5  // указывает на 3
```

Обычный итератор `rit.base()`:

```
1 2 3 -> 4 5  // указывает на 4
```

`std::reverse_iterator`

Важно понимать, что элемент, на который указывает `reverse_iterator` фактически отличается от того, на который указывает обычный итератор.

Это устроено так, потому что контейнеры не обязаны поддерживать элемент "перед begin". К тому же это упрощает реализацию методов `rbegin` и `rend`:

```
std::reverse_iterator<iterator> rbegin() {
  return std::make_reverse_iterator(end());  // <-- последний элемент
  // end:    1 2 3 ... n ->
  // rbegin: 1 2 3 ... n <-
}

std::reverse_iterator<iterator> rend() {
  return std::make_reverse_iterator(begin());  // <-- элемент перед первым
  // begin: -> 1 2 3 ... n
  // rend:  <- 1 2 3 ... n
}
```

`reverse_iterator`, `const_reverse_iterator`

В стандартных контейнерах типы `reverse_iterator` и `const_reverse_iterator` определены следующим образом:

```
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
```

`std::move_iterator`

`std::move_iterator`

Допустим, хотим, чтобы при разыменовании итератора возвращалась не ссылка на объект, а `rvalue` ссылка. Это может быть нужно, например, для того, чтобы переместить элементы одного буфера в другой:

```
std::vector<std::vector<int>> vv;
// ...
// Теперь нужен список из тех же векторов, старый вектор больше не понадобится
std::list<std::vector<int>> l(vv.begin(), vv.end());  // <- копирование
vv.clear();
```

Для этого достаточно воспользоваться адаптером `std::move_iterator`:

```
std::vector<std::vector<int>> vv;
// ...
std::move_iterator<std::vector<std::vector<int>>::iterator> begin(vv.begin());
std::move_iterator<std::vector<std::vector<int>>::iterator> end(vv.end());
std::list<std::vector<int>> l(begin, end);  // <- перемещение
vv.clear();
```

`std::move_iterator`: `std::make_move_iterator`

для того, чтобы не писать длинное имя типа можно воспользоваться функцией `std::make_move_iterator`:

```
std::vector<std::vector<int>> vv;
// ...
auto begin = std::make_move_iterator(vv.begin());
auto end = std::make_move_iterator(vv.end());
std::list<std::vector<int>> l(begin, end);  // <- перемещение
vv.clear();
```

`std::move_iterator`: как устроен

Устроен очень просто: в `operator*` возвращает `std::move(*it)`, где `it` - итератор, от которого он был создан (`base()`).

Остальные методы эквивалентны обычному итератору.

Потоковые итераторы

Потоковые итераторы

Потоковые итераторы позволяют работать с потоками ввода и вывода так, как если бы это были обычные последовательности элементов.

Представлены классами `std::istream_iterator`, `std::istream_iterator`:

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(std::cin),
          std::istream_iterator(),
          std::back_inserter(v));
// ... полезная работа
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
```

Разыменование первого итератора возвращает считанный элемент, а его инкремент считывает следующий.

Присваивание второму итератору приводит к записи в поток. Остальные операции ничего не делают.