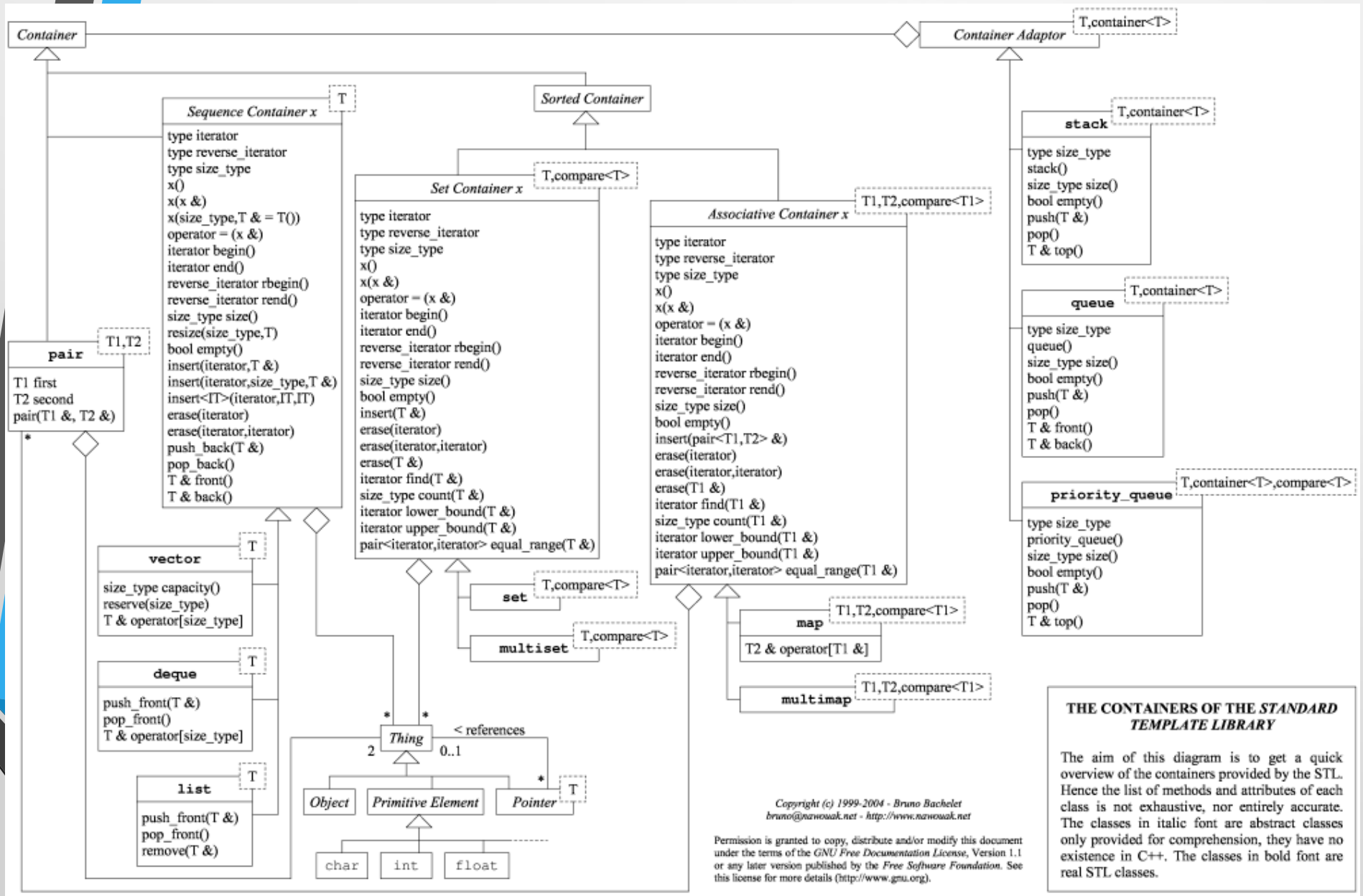
Лекция №5

Кросс-платформенное программирование

STL – это просто =), наверное



Сокращенное строение библиотеки STL

Библиотека STL

Standard Template Library (STL, Александр Степанов и Менг Ли, Hewlett-Packard Lab) - надстройка над C++.

Задачи:

- Упростить работу с C++
- Сделать ее «комфортной».

Главная идея **STL** - уменьшение зависимости от стандартных библиотек **C++**. Основная проблема стандартных библиотек - их тесная связь с данными, что делает эти библиотеки неудобными для работы с типами данных пользователя. **STL** позволяет работать с любыми типами данных и производить над ними операции.

- **STL** отделяет структуры данных от алгоритмов, которые с ними работают.

С 1994 года STL стала частью официального стандарта языка C++.

Основные компоненты STL

Контейнер (англ. *container*) — объект, который содержит другие объекты.

Итератор (англ. *iterator*) — объект-указатель на контейнер.

Итератор циклически опрашивает содержимое контейнера.

Алгоритм (англ. *algorithm*) — вычислительная процедура для обработки контейнеров.

Виды алгоритмов: инициализация, сортировка, поиск, преобразование содержимого контейнера.

Основные компоненты STL

Разделение алгоритмов и данных позволяет уменьшить количество компонентов.

Например, вместо написания функции поиска элемента для каждого вида контейнера мы обеспечиваем единственную версию, которая работает с каждым из них, пока удовлетворяется основной набор требований.

Возможности STL

- классы *string и wstring* реализующих динамические строки (с однобайтовыми и двубайтовыми символами);
- класс complex реализующий комплексные числа;
- классы по локализации приложений;
- потоки ввода/вывода для файлов, консоли и строк;
- классы обработки исключений;
- *итераторы* - сходные по функциональности с указателями объекты, используемые для обработки элементов контейнерных типов;
- контейнерные классы - классы по управлению множеством элементов одного типа, как
  - *vector* - динамический массив;
  - *list* - список;
  - *queue,deque* - очередь;
  - *stack* - стек;
  - *map, multimap* - отображения (ассоциативные массивы);
  - *set* - множество;
- *алгоритмы* - шаблоны функций для обработки элементов массивов и контейнерных классов;
- различные вспомогательные классы
  - *функциональные объекты* - классы для которых перегружена операция (), используется в алгоритмах;
  - *pair* - класс реализующий пару значений, используемый с отображениями;
  - *auto_ptr* - простой "умный" указатель.

Контейнеры

Последовательные контейнеры

- **vector** – динамический массив
- **list –** двусвязный список
- **deque –** дэк

Ассоциативные контейнеры

- **set** – множество уникальных элементов
- **multiset** – множество необязательно уникальных элементов
- **map** – упорядоченный ассоциативный массив пар элементов (ключ/значение)
- **multimap –** как map, но можно хранить повторы

Контейнеры

Псевдоконтейнеры

stack – стек, реализация LIFO**.** Добавление и удаление элементов осуществляется с одного конца.

queue – очередь, реализация FIFO. С одного конца можно добавлять элементы, а с другого — вынимать.

priority_queue - очередь с приоритетом. Элемент с наивысшим проиоритетом всегда стоит на первом месте.

Очередь с приоритетом

Очередь с приоритетом (англ. *priority queue*) — структура данных, поддерживающая три операции:

- **InsertWithPriority**: добавить в очередь элемент с назначенным приоритетом
- **GetNext**: извлечь из очереди и вернуть элемент с минимальным приоритетом (другие названия «PopElement(Off)» или «GetMinimum»)
- **PeekAtNext** (необязательная операция): просмотреть элемент с наивысшим приоритетом без извлечения

Контейнер vector

vector - динамический массив произвольного доступа с автоматическим изменением размера при добавлении/удалении элемента.

#include <vector> // подключение библиотеки
vector<*тип*> *имя_переменной*; // объявление

Контейнер vector

Основные функции в классе vector:

- begin() – итератор начального элемента
- end() – итератор последнего элемента
- clear() – удаление всех элементов вектора
- resize(*новая_длина*) // изменение размера вектора
- size() // получение текущего размера вектора
- push_back(T &val) // добавляет эл-нт в конец вектора
- insert(iterator i, T &val) // добавляет эл-нт внутрь вектора
- [] // обращение к элементу вектора

Контейнер vector

Пример

```
vector<int> vec;
for (unsigned i = 0; i < 5; i++)
    vec.push_back(i*i);

for (unsigned i = 0; i < vec.size(); i++)
    cout << vec[i] << " ";

vec.resize(10); // теперь размер 10
…
vec.clear();
```

Вывод:
0 1 4 9 16

Контейнер vector

Вектор может быть многомерным.

Пример:

```
vector< vector<int> > vec_matrix;
vec_matrix.resize(5); // у матрицы 5 строк
for (unsigned i = 0; i < vec_matrix.size(); i++)
      vec_matrix[i].resize(6); // 6 столбцов
```

Вектор может быть не только числовым.

```
vector<string> vec_string; // вектор строк
```

Действуют операторы присваивания и сравнения.

```
vec1 = vec2;
If ( vec1 < vec2 ) …
```

Контейнер list

list – двусвязный список Поиск перебором медленнее, чем у вектора из-за большего времени доступа к элементу. Вставка и удаление производятся быстрее.

#include <list> // подключение библиотеки
Некоторые функции как у vector. Но есть функции объединения (merge) двух списков и сортировки списка (sort).

Пример:

```
list<int> lst;
lst.resize(5);
for (unsigned i = 0; i < 5; i++)
    lst.push_back(i*i);
lst.sort();
```

Методы работы со стеком

void push(<type>) // добавление элемента в стек.

void pop() // удаляет элемент с вершины стека.

<type> top() // возвращает элемент с вершины стека.

unsigned int size() // определяет размер стека (количество элементов).

bool empty() // возвращает истину, если стек пуст

Стек (Stack)

Чтобы использовать стек, необходимо подключить библиотеку <stack>. Приведем пример программы

```
#include <stack>
#include <stdio.h>
using namespace std;
int main()
{
    stack <int> S;
    S.push(8);
    S.push(7);
    int x = S.size(); //x==2
    while (!S.empty()) {
        printf("%d ", S.top());
        S.pop();
    }
return 0;
}
```

Контейнер map

Позволяет хранить пары вида «(ключ, значение)».

Поддерживает операции добавления пары, а также поиска и удаления пары по ключу:

- INSERT(ключ, значение)
- FIND(ключ)
- REMOVE(ключ)

Ключи должны быть уникальны. Порядок следования элементов определяется ключами.

Контейнер map

#include <map> // подключение библиотеки

Пример.

```
map<char, int> m;
char ch;
for (int i = 0; i < 10; i++)
    m.insert(pair<char, int>('A' + i, i ) );
cout << "Enter ch";
cin >> ch;
map<char,int> :: Iterator p;
p = m.find(ch);
if ( p != m.end() )
    cout << p.second();
else
    cout << "Ключа нет";
```

Итераторы

Итераторы - это обобщение указателей, которые позволяют программисту работать с различными структурами данных (контейнерами) единообразным способом.

Итераторы - это объекты, которые имеют оператор *, возвращающий значение некоторого класса или встроенного типа *T*, называемого *значимым типом* (*value type*) итератора.

Итераторы

Точно также, как обычный указатель на массив гарантирует, что имеется значение указателя, указывающего за последний элемент массива, так и для любого типа итератора имеется значение итератора, который указывает за последний элемент соответствующего контейнера.

Эти значения называются *законечными* (*past-the-end*) значениями.

Итераторы

Объявление итератора
тип_контейнера :: iterator имя_переменной;

Пример
```
list<int> :: iterator lst_it; // итератор
list<int> lst;

for (lst_it = lst.begin(); lst_it != lst.end(); lst_it++ )
    cout << *lst_it << " "; // выводим эл-т списка

list< list<int> > lst;
list< list<int> > :: iterator lst_it; // итератор
for (lst_it = lst.begin(); lst_it != lst.end(); lst_it++ )
    cout << (*lst_it).size << " "; // эл-т списка = список
```

Итераторы

Когда пользователь хочет объединить вектор и список (оба - шаблонные классы в библиотеке) и поместить результат в заново распределённую неинициализированную память, то это может быть выполнено так:

```
vector<Employee> a;
list<Employee> b; ...
Employee* c = allocate(a.size() + b.size(), (Employee*) 0);
merge(a.begin(), a.end(), b.begin(), b.end(),
 c.begin() );
```

где *begin()* и *end()* - функции-члены контейнеров, которые возвращают правильные типы итераторов.

Алгоритмы в STL

Библиотека расширяет основные средства C++ последовательным способом, так что программисту на легко начать пользоваться библиотекой.

Все алгоритмы отделены от деталей реализации структур данных и используют в качестве параметров типы итераторов.

Алгоритмы могут работать с определяемыми пользователем структурами данных, если эти структуры данных имеют типы итераторов, удовлетворяющие предположениям в алгоритмах.

Алгоритмы в STL

Не меняющие последовательность операции

- Операции с каждым элементом (For each)
- Найти (Find)
- Найти рядом (Adjacent find)
- Подсчет (Count)
- Отличие (Mismatch)
- Сравнение на равенство (Equal)
- Поиск подпоследовательности (Search)

Алгоритмы в STL

Меняющие последовательность операции

- Копировать (Copy)
- Обменять (Swap)
- Преобразовать (Transform)
- Заменить (Replace)
- Заполнить (Fill)
- Породить (Generate)
- Удалить (Remove)
- Убрать повторы (Unique)
- Расположить в обратном порядке (Reverse)
- Переместить по кругу (Rotate)
- Перетасовать (Random shuffle)
- Разделить (Partitions)

Алгоритмы в STL

Операции сортировки и отношения

Операции над множеством для сортированных структур

…

Алгоритмы в STL

Микро-алгоритмы

swap(T &a, T &b)

Меняет местами значения двух элементов.

iter_swap(It p, It q)

Меняет местами значения элементов, на которые указывают итераторы.

max(const T &a,const T &b)

Возвращает максимальный элемент.

min(const T &a,const T &b)

Возвращает минимальный элемент.

У этих алгоритмов есть версии с тремя параметрами. Третий параметр принимает бинарный предикат, задающий упорядоченность объектов.

Алгоритмы в STL

Алгоритмы, не модифицирующие последовательности

size_t count(It p, It q, const T &x)
Возвращает, сколько раз элемент со значением x входит в последовательность, заданную итераторами p и q.

size_t count_if(It p, It q, Pr pred)
Возвращает, сколько раз предикат pred возвращает значение true.
Например, count_if(p, q, divides_by(8)) вернет, сколько элементов кратно 8;

Алгоритмы в STL

Алгоритмы типа find

find(It p, It q, const T &x)
Возвращает итератор на первое вхождение элемента x в последовательность, заданную итераторами p и q.

find_if(It p, It q, Pr pred)
Возвращает итератор на первый элемент, для которого предикат pred вернул значение true.

find_first_of(It p, It q, Itr i, Itr j)
Возвращает итератор на первое вхождение любого элемента из последовательности, заданной итераторами i и j, в последовательность, заданную итераторами p и q. Последовательности могут быть разных типов (например std::vector и std::list).

Алгоритмы в STL

Алгоритмы типа find

min_element(It p, It q)
Возвращает итератор на минимальный элемент последовательности.

max_element(It p, It q)
Возвращает итератор на максимальный элемент последовательности.

equal(It p, It q, Itr i)
Сравнивает две последовательности на эквивалентность. Вторая последовательность задается одним итератором, так как последовательности должны быть одинаковой длины. Если вторая короче, то undefined behaviour.

Алгоритмы в STL

Алгоритмы типа find

pair <It, Itr> mismach(It p, It q, Itr i)

Возвращает пару итераторов, указывающую на первое несовпадение последовательностей. F

for_each(It p, It q, F func)

Для каждого элемента последовательности применяет функтор func. Возвращаемое значение функтора после каждого применения игнорируется. Возвращает функтор func после его применения ко всем элементам.

Алгоритмы в STL

Алгоритмы типа find

bool binary_search(It p, It q, const T &x)

Возвращает true, если в упорядоченной последовательности есть элемент, значение которого равно x, false в противном случае.

Если хотим получить итератор на элемент со значением x, то нужно использовать алгоритмы lower_bound(It p, It q, const T &x), upper_bound(It p, It q, const T &x), equal_range(It p, It q, const T &x), которые выполняют то же, что и одноименные методы для контейнера std::set. Эти алгоритмы работают за линейное время на BiDi итераторах и за логарифмическое время на RA итераторах.

Алгоритмы в STL

Модифицирующие алгоритмы

fill(It p, It q, const T &x), fill_n(It p, Size n, const T &x)
Заполняют последовательность значениями, равными значению x.

generate(It p, It q, F gen), generate_n(It p, Size n, F gen)
Заполняют последовательность значениями, сгенерированными функтором gen (например, генератором случайных чисел).

Алгоритмы в STL

Модифицирующие алгоритмы

random_shuffle(It p, It q), random_shuffle(It p, It q, F &rand)

Перемешивает элементы в случайном порядке: меняет местами каждый элемент с элементом, номер которого выбирается случайно. Третьим параметром можно задать функтор, который будет выбирать этот случайный номер. Можно передавать генератор случайных чисел, но распределение должно быть равномерным (каждая перестановка должна генерироваться с вероятностью 1/n!, а это совсем не то же самое, что каждый элемент окажется на i-м месте с вероятностью 1/n). Требует RA итераторов.

Алгоритмы в STL

Модифицирующие алгоритмы

copy(It p, It q, Itr out)

Копирует значения элементов последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out.

copy_backward(It p, It q, Itr out) Копирует элементы последовательности, заданной итераторами p и q, в последовательность, заканчивающуюся итератором out. Итераторы должны быть BiDi.

Алгоритмы в STL

Модифицирующие алгоритмы

remove_copy(It p, It q, Itr out, const T &x)
Копирует значения элементов из последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out, за исключением элементов, значения которых равны значению x.

remove_copy_if(It p, It q, Itr out, Pr pred)
Копирует значения элементов из последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out, за исключением элементов, для которых предикат pred возвращает значение true.

Алгоритмы в STL

Модифицирующие алгоритмы

reverse(It p, It q)
Переставляет элементы в обратном порядке.

reverse_copy(It p, It q, Itr out)
Копирует значения элементов в обратном порядке.

rotate(It p, It middle, It q)
Сдвигает элементы последовательности так, что элемент, на который указывает итератор middle становится первым.

Алгоритмы в STL

Модифицирующие алгоритмы

swap_ranges(It p, It q, Itr i)

Меняет местами элементы последовательности, заданной итераторами p и q, с соответствующими элементами последовательности, начинающейся с и итератора out.

remove(It p, It q, const T &x)

Удаляет из последовательности элементы, значения которых совпадают по значению с x. Возвращает итератор на новый конец последовательности.

Например:

Алгоритмы в STL

Модифицирующие алгоритмы

unique(It p, It q), unique(It p, It q, Pr pred)

Удаляет одинаковые подряд идущие элементы, оставляя только по одному элементу для каждого значения. Элементы последовательности должны быть отсортированы. Работает аналогично алгоритмам remove и remove_if, оставляя в начале только уникальные элементы, а в конце - то, что осталось. В качестве третьего параметра можно передавать предикат, сравнивающий два элемента и возвращающий true, если элементы равны, и false в противном случае.

Алгоритмы в STL

Модифицирующие алгоритмы

transform(It p, It q, Itr out, F func)

К каждому элементу входящей последовательности применяет функтор func и записывает результат в последовательность, начинающуюся с итератора out.

transform(It p, It q, Itr i, Iter out, F func)

Применяет бинарный функтор func к каждой паре элементов из двух входящих последовательностей и записывает результат в результирующую последовательность.

Алгоритмы в STL

Модифицирующие алгоритмы

accumulate(It p, It q, T i, F func)

Последовательно применяет бинарный функтор func к парам (i, *p++), где i - некоторое начальное значение, которое затем каждый раз заменяется значением, которое возвращает функтор. Функтор должен возвращать значение типа T.

Реализация этого алгоритма выглядит примерно следующим образом: while (p != q) { i = func(i, *(p++)); } return i; Например, если в качестве i передать 0, а в качестве func - функтор, вычисляющий сумму, то посчитаем сумму элементов последовательности. Если в качестве i передать 1, а в качестве func - функтор, вычисляющий произведение, то получим произведение элементов и т.д.

Алгоритмы в STL

Модифицирующие алгоритмы

sort(It p, It q), sort(It p, It q, Pr pred)

Сортирует элементы последовательности в порядке возрастания. stable_sort(It p, It q), stable_sort(It p, It q, Pr pred)

Сортирует элементы, сохраняя порядок элементов с одинаковыми значениями относительно друг друга. Эти алгоритмы требуют RA итераторов, поэтому на списке работать не будут. Но у списка есть собственные функции члены sort, stable_sort.

Алгоритмы в STL

Модифицирующие алгоритмы

void nth_element(It p, It nth, It q), void nth_element(It p, It q, It nth, Pr pred)

Позволяет получить n-й по порядку элемент (n-й по счету, как если бы массив был отсортирован), переставляя элементы таким образом, что все элементы до него меньше, либо равны ему, а элементы после - больше, либо равны ему. partition(It p, It q, Pr pred)

Переставляет элементы последовательности таким образом, что все элементы, для которых предикат вернул true, предшествуют тем, для которых он вернул false. Возвращает итератор на первый элемент из второй группы.

Алгоритмы в STL

Модифицирующие алгоритмы

void partial_sort(It p, It middle, It q), void partial_sort(It p, It middle, It q, Pr pred)

Переставляет элементы последовательности так, что элементы межу итераторами p и q располагаются в том порядке, как если бы последовательность была отсортирована, а элементы в оставшейся части - в произвольном порядке. То есть получаем часть отсортированной последовательности (не то же самое, что отсортированную часть). merge(It p, It q, Itr i, Itr j, Iter out), merge(It p, It q, Itr i, Itr j, Iter out, Pr pred)

Сортирует две последовательности слиянием.

Алгоритмы в STL

Библиотека содержит шаблонную функцию *merge* (слияние). Когда пользователю нужно два массива *a* и *b* объединить в *c*, то это может быть выполнено так:

```
int a[1000];
int b[2000];
int c[3000];
 ...
merge(a, a+1000, b, b+2000, c);
```

Алгоритмы в STL

sort() сортирует элементы в диапазоне *[first, last)*.

void sort(RandomAccessIterator first,
RandomAccessIterator last);

random_shuffle переставляет элементы в диапазоне *[first, last)* с равномерным распределением.

void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last);

Алгоритмы в STL

Пример

```
#include <vector>
#include <algorithm>
…
list<int> lst;
list<int> :: iterator lst_it;
for ( unsigned i=0; i < 10; i++ )
      lst.push_back(i);

random_shuffle(lst.begin(), lst.end());
for ( lst_it = lst.begin(); lst_it != lst.end(); lst_it++ )
     cout << *lst_it;
sort(lst.begin(); lst.end());
for ( lst_it = lst.begin(), lst_it != vec.end(); lst_it++ )
     cout << *lst_it;
```

Пример accumulate

```
#include <iostream>
#include <vector>
#include <numeric>
#include <string>
#include <functional>
int main()
{
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int sum = std::accumulate(v.begin(), v.end(), 0);

    int product = std::accumulate(v.begin(), v.end(), 1,
std::multiplies<int>());

    std::string s = std::accumulate(std::begin(v),
std::end(v), std::string{},
                [](const std::string& a, int b) {
                return a.empty() ? std::to_string(b):
                        a + '-' + std::to_string(b);
                        });

    std::cout << "sum: " << sum << '\n'
              << "product: " << product << '\n'
              << "dash-separated string: " << s << '\n';
}
```

Output

```
sum: 55
product: 3628800
dash-separated string: 1-2-3-4-5-6-7-8-9-10
```

Класс auto_ptr

Ограничения класса auto_ptr (простой "умный" указатель) :

- объектом может владеть только один указатель,
- объектом не может быть массив,
- нельзя использовать адресную арифметику.

Единственное назначение этого класса - *автоматизировать уничтожение* выделенной ранее памяти.

Данный класс используется, когда время существование выделенного объекта можно ограничить определенным блоком.

Делая код более безопасным, данные классы не наносят ущерб размеру или скорости программы.

Пример

```
1. #include <memory> // объявление шаблона класса auto_ptr
2. #include <iostream>
3. using namespace std;

4. // Внутри функции мы выделяем память для объекта типа int
5. // но не освобождаем ее явно оператором delete.
6. // Это делается автоматически.
7. void main(void)
8. {
9.  auto_ptr<int> aptr(new int(20));
10. auto_ptr<int> aptr2;
11. cout<<"*aptr="<<*aptr<<endl;
12. aptr2=aptr; // теперь aptr не владеет никаким объектом
13. cout<<"*aptr2="<<*aptr2<<endl;
14.}
```

Пара (Pair)

Пара, фактически является шаблонной структурой, которая содержит два поля (возможно, разных типов), называются они first и second. Для того чтобы использовать pair, необходимо подключить библиотеку <utility>

Пусть, например, мы хотим создать пару из целого и вещественного числа. Тогда ее создание будет выглядеть следующим образом:

pair <int, double> p;

Теперь мы можем обращаться к p точно так же, как к обычной структуре, например, так:

p.first = 5; p.second = 3.1415;

Пары одинакового типа можно присваивать друг другу. Пары используются в ассоциативных контейнерах (о них будет сказано позже). Поля пары могут быть не только элементарного, но и составного типа, например, опять же парой. Для примера, приведем реализацию структуры, хранящей дату с использованием пар (год, месяц, день):

Пара (Pair)

pair <int, pair <int, int> > date1, date2;

Стоит заметить, что > разделены пробелом, если не разделять их, то эта запись будет интерпретироваться как оператор сдвига вправо >>, что приведет к ошибке при компиляции.

Обращение к полям будет выглядеть так:

int year = date1.first;

int month = date1.second.first;

int day = date1.second.second;

Крайне полезное свойство состоит в том, что пары можно сравнивать. При этом сравнение идет слева направо. Т.е. для нашей даты сначала сравнятся годы, при равных годах сравнятся месяцы и т.д. Это очень удобно использовать при сортировке.