

Стандартная библиотека шаблонов C++ (STL)

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Пространства имен (**namespace**)

- Контейнер для имен, чтобы избежать пересечений:

```
namespace parking {  
    class Gate { /* Ворота на парковке. */ };  
}  
namespace schematics {  
    class Gate { /* Логический вентиль в электронике. */ };  
}  
parking :: Gate east_gate;  
schematics :: Gate sign_flipper;
```

- Все стандартные компоненты — в пространстве имен **std**.
 using namespace std; в **sdt.h** позволяет не писать **std :: cin** и т. п.
 - В реальном коде так делать не следует!
- Класс или структура тоже создают свое пространство имен,
 - отсюда и оператор **::** при работе с ними.

Состав STL

- Контейнеры
- Алгоритмы
- Итераторы
- Ввод и вывод
 - потоки (`<iostream>`, `<sstream>`, `<fstream>`);
 - форматирование (`<iomanip>`).
- Вспомогательные средства
 - стандартные классы исключений (`<stdexcept>`);
 - определения `size_t`, `uint16_t` и т. д. (`<stdint>`);
 - «умные указатели» (`<memory>`).
- Специальные библиотеки
 - многопоточное программирование, работа с датой и временем...
 - сейчас (2014—2017 г. г.) идет активное развитие этой части:
 - скоро: работа с файлами и каталогами, с сетью...

Линейные контейнеры STL

- `vector<T>` — динамический массив:
 - доступ по индексу за $O(1)$,
вставка и удаление элементов за $O(N)$, с концов быстрее;
 - самый универсальный.
- `array<T, N>` — статический массив размера N :
 - N должно быть известно при компиляции;
 - не может изменять размер;
 - не выделяет память динамически, «создается» быстрее вектора.
- `list<T>` — двусвязный список:
 - вставка, удаление, обмен элементов за $O(1)$;
 - доступ по индексу за $O(N)$, реально еще медленнее.
- `forward_list<T>` — односвязный список:
 - меньше накладные расходы узлов;
 - не все операции доступны;
 - введен для тонких оптимизаций.

Ассоциативные контейнеры STL

- Упорядоченные:

основаны на древовидных структурах данных, требуют сравнимости ключей (элементов).

- Основные операции за $O(\log N)$.
- `map < K, V >` — ассоциативный массив,
- `set < T >` — множество уникальных элементов.

- Неупорядоченные:

основаны на хэш-таблицах, требуют возможности вычисления хэша ключа (элемента) через `hash < T >()`.


- Поиск за $O(1)$, в худшем случае — $O(N)$.
- `unordered_map < K, V >`, `unordered_set < T >`

- Допускающие повторяющиеся ключи (элементы):

- `multimap < K, V >`, `multiset < T >`
- `unordered_multimap < K, V >`, `unordered_multiset < T >`

Адаптеры к контейнерам STL

«Стек — это _____, добавлять и удалять из которого можно только последний элемент.

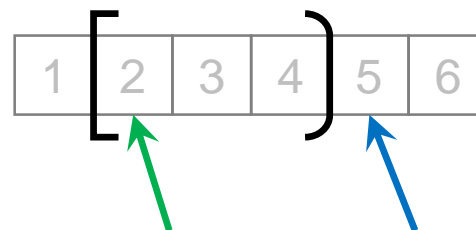


вектор список дек

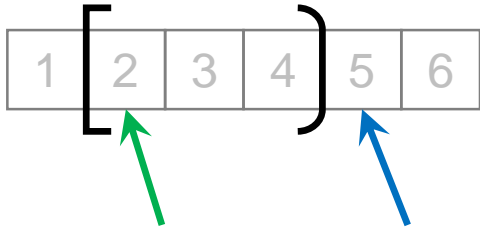
- Предоставляют одинаковый ограниченный интерфейс к последовательным (линейным) контейнерам.
 - Не хранят элементы (но хранят контейнер с ними).
 - Позволяют выбрать контейнер при объявлении.
- `stack<T>` \Leftrightarrow `stack<T, vector<T>>` — стек.
- `queue<T>` — очередь.
- `priority_queue<T>` — очередь с приоритетами:
 - добавление за $\mathcal{O}(\log N)$ с указанием приоритета элемента;
 - извлечение за $\mathcal{O}(1)$ в порядке убывания приоритетов.

Обработка контейнеров

- У некоторых контейнеров нет `find()` и т. п.
- Можно сделать внешнюю функцию,
- но что туда передать?
 - Контейнер?
 - Можно, но не гибко.
 - Часть контейнера?
 - Копирование нежелательно!
 - Индексы?
(Как в «быстрой» сортировке.)
 - Не для всех контейнеров подходит.
 - Как-то представить стрелочки.



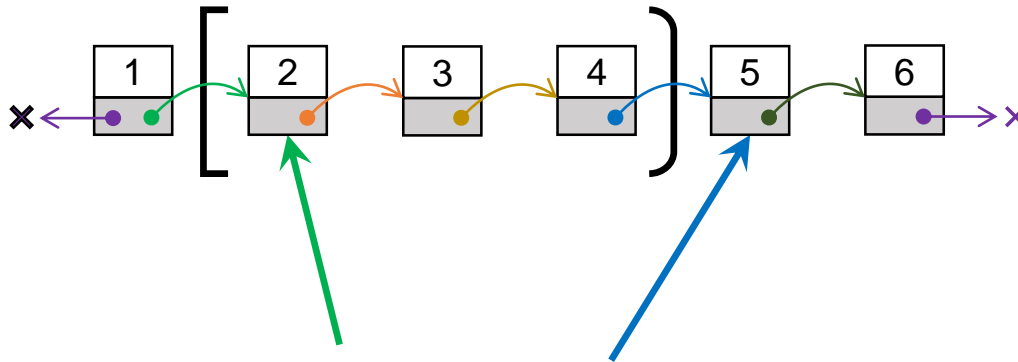
Итераторы (iterators)



Arrow find (Arrow from, Arrow to, Type what)

```
{
    Arrow current = from;
    while (current != to)
    {
        if (what == *current)
            return current;
        current = next(current);
    }
    return to;
}
```

- 1) Нужно проверять, что стрелки указывают на разные элементы.
- 2) Нужно получать значение, хранимое элементом, на который указывают.
- 3) Нужно перемещать стрелку к следующему элементу.

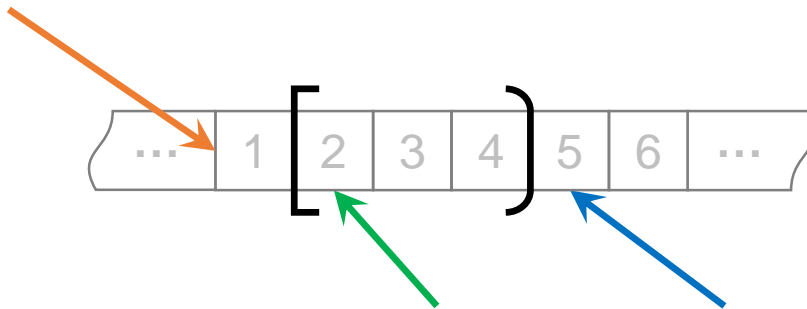


`Node* find (Node* from, Node* to, Type what)`

```

{
    Node* current = from;
    while (current != to)
    {
        if (what == current->value)
            return current;
        current = current->next;
    }
    return to;
}
  
```

&values[0] vector < **double** > values { 1, 2, 3, 4, 5, 6 };



double* find(double* from, double* to, Type what)

```
{
    double* current = from;
    while (current != to)
    {
        if (what == *current)
            return current;
        current = current + 1;
    }
    return to;
}
```

Перемещение к следующему **double** в памяти (на 8 байт вперед):

&values[1] == &values[0] + 1
&values[2] == &values[1] + 1
и т. д.

Итераторы контейнеров STL

```
vector<double> data { ... };  
vector<double>::iterator current = data.begin();  
while ( current != data.end() )  
{  
    cout << *current;  
    current = next(current); // ++current;  
}  
  
vector<double>::iterator it = find ( data.begin(), data.end(), 42 );  
if ( it != data.end() ) {  
    // Элемент со значением 42 найден.  
}
```

Инвалидация итераторов

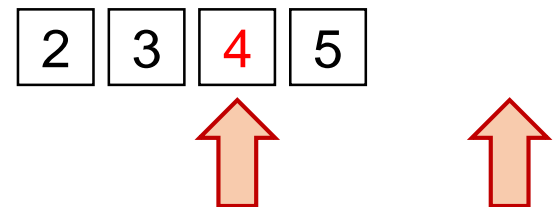
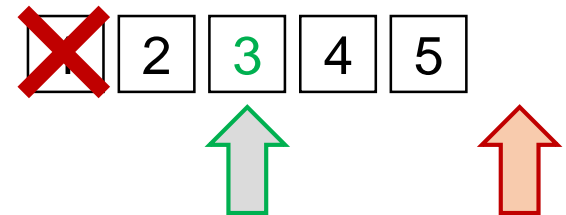
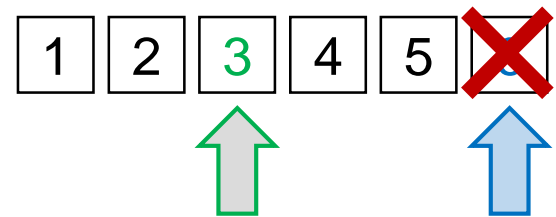
- После *определенных* действий над *некоторыми* контейнерами полученные ранее итераторы становятся недействительными.
- Нужно знать, когда это происходит.
 - Написано в стандарте (документации).
 - Исходя из знаний о структурах данных.

× **for** (**auto** it = xs.begin(); it != xs.end(); ++it)
 if (...) xs.erase(it);

✓ **for** (...)
 if (...) it = xs.erase(it);
 else ++it;

✓ remove_if (xs.begin(), xs.end(), ...);

vector<T>



Функции STL для итераторов

- **auto** successive = next (current), previous = prev (current);
- **auto** advanced = advance (current, 42);
- size_t elements_between = distance (first, last);
 - Для итераторов одного контейнера;
 - last должен следовать после first.
- begin (xs) == xs .begin(), end(), cbegin(), cend()
- iter_swap (a, b);
 - Меняет местами элементы, с которыми связаны итераторы.
 - Одного и того же контейнера!
 - Позиции итераторов a и b не изменяются, но изменяется *a и *b.
 - Может быть реализовано эффективнее swap().

Иерархия итераторов STL

- **Input:**
 - только чтение;
 - переход к следующему.
- **Forward = Input +:**
 - возможно несколько проходов.
- **Bidirectional = Forward +:**
 - переход к предыдущему.
- **Random-access = Bidirectional +:**
 - т. н. произвольный доступ к элементам, отстоящим от итератора на любое расстояние.
- **Output (вне иерархии):**
 - запись;
 - переход к следующему.

Специализированные итераторы

- По элементам, считываемым из потока или записываемым в него (*`*stream_iterator`*).
- Вставляющий элементы в контейнер — в конец, в начало (*`*inserter_iterator`*)
- Для прохода в обратном направлении от заданного итератора (*`reverse_iterator`*).

Применение алгоритмов с итераторами

- Пройти по записям в файле, добавляя их в вектор:

```
vector< Order > orders;  
ifstream database ( "orders.dat" );  
copy ( istream_iterator ( database), istream_iterator(),  
       back_inserter ( orders ) );
```
- Вывести нарастающие итоги на отдельных строках:

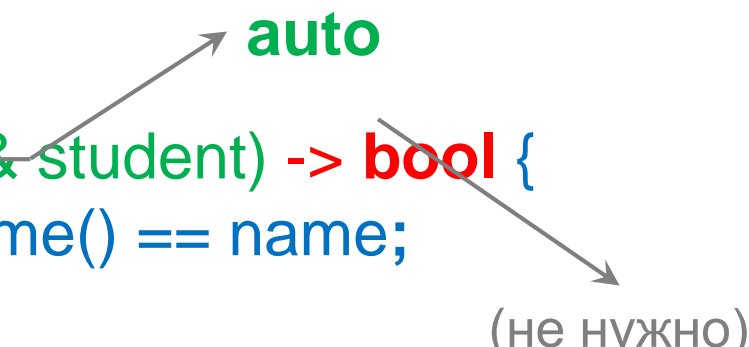
```
partial_sum(  
    begin(xs), end(xs), ostream_iterator ( cout, "\n" ) );
```
- Удалить наименьший элемент:

```
auto where = min_element ( xs.begin(), xs.end() );  
if ( where != xs.end() )  
    xs.erase ( where );
```

Замыкания (лямбда-функции)

- `find(xs.begin(), xs.end(), value);`
 - Как найти не `value`, а все значения, удовлетворяющие условию?
- Цель: `find_if(..., условие);`
- Условие:
 - Можно вызвать как функцию `bool satisfies(T const& item);`
 - Обыкновенная функция.
 - Объект с `bool operator () (T const& item)` — т. н. функциональный объект (функтор).
 - Может зависеть от параметров:
 - Найти студента с заданным именем.

Замыкания (λ-функции)

- **string** name = "Василий";
find_if(xs.begin(), xs.end(),
 [&name] (~~Student const& student~~) -> **bool** {
 return student.get_name() == name;
 });
- 
- (не нужно)

- Список захвата (capture list):

[] в λ-функции можно использовать только параметры;

[&name, ...] переменная **name** доступна в λ как ссылка;

- для организации выходного параметра;
- во избежание копирования.

[&] все локальные переменные доступны в λ как ссылки;

[name, ...] в λ доступна копия переменной **name**;

[=] в λ доступны копии всех локальных переменных.

Лямбда-функции и типы

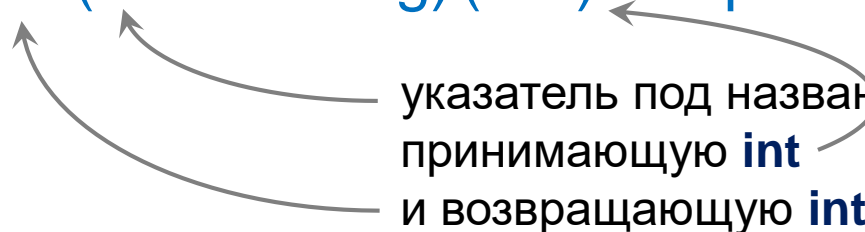
- `int find_if (vector < double > xs, ??? satisfies);`
- `#include <functional>`
`int find_if (vector < double > xs, function < bool (double) > satisfies)`
`{`
 `for (int i = 0; i < xs.size(); ++i)`
 `if (satisfies(xs[i])) return i;`
 `return -1;`
`}`
- `template < typename T >`
`int find_if (vector < double > xs, const T& satisfies) {`
 `for (int i = 0; i < xs.size(); ++i)`
 `if (satisfies(xs[i])) return i;`
 `return -1;`
`}`

Указатель на функцию и функция обратного вызова (callback)

- `int square(int x) { return x*x; }`

- `auto something = square;`

- `// int (*something)(int) = square;`



указатель под названием `something` на функцию,
принимающую `int`
и возвращающую `int`

- `something(2); // 4`

- Широко применяются в C, где нет λ -функций.

- `qsort()` — быстрая сортировка, `bsearch()` — двоичный поиск.
 - системные и прикладные библиотеки.

Литература к лекции

- *Programming Principles and Practices Using C++:*
 - часть III:
 - глава 20 — линейные контейнеры, итераторы;
 - глава 21 — ассоциативные контейнеры, алгоритмы;
 - часть V — стандартная библиотека и STL.
- *C++ Primer:*
 - часть II:
 - глава 8 — ввод и вывод;
 - глава 9, 11 — контейнеры;
 - глава 10 — алгоритмы.
- ***C++ Reference***