Программирование на C++ и Python

Лекция 3. Стандартная библиотека *C++*

Воробьев Виталий Сергеевич (ИЯФ, НГУ)

9 октября 2019, Новосибирск

Стандартная библиотека

isocpp.org

Строки
- string
- regex

Библиотеки C
- [21 файл]

Потоки ввода-вывода
- ios
- iostream
- fstream
- iomanip
- sstream
- iosfwd
- ostream
- streambuf

Контейнеры STL
- set
- list
- array
- map
- bitset
- vector
- stack
- queue
- forward_list
- unordered_set
- unordered_map

Численные библиотеки
- complex
- random
- valarray
- numeric

Вспомогательные
- exception
- limits
- new
- typeinfo

Общие
- algorithm
- chrono
- memory
- functional
- iterator
- stdexcept
- tuple
- utility

Многопоточность
- thread
- mutex
- shared_mutex
- condition_variable
- future

Локализация
- locale
- codesvt

(Полный список смотрите на cppreference.com/w/cpp/header)

std::string

- Класс для хранения 8-битовых символьных строк
- Поддерживает множество операций (смотрите документацию)
- Поддерживают операторы сравнения (лексикографический порядок) и операторы ввода-вывода
- Имеет инструменты для поиска внутри строки

```
#include <string>
#include <iostream>
using namespace std;

int main() {
  string a("Hello, ");
  string b("world");
  string c = a + b + "!";  // Hello, world!
  cout << c.size() << endl;  // 13
  string d = c.substr(7, 5);  // world
  // Hello, Mike!
  c.replace(c.find("world"), 5, "Mike");

  int n = stoi("456");
  double x = stod("5.654");
  string e = "pi equals " + to_string(3.1415);
}
```

Как вернуть несколько значений?

```cpp
#include <utility>
#include <tuple>
#include <string>
using namespace std;

pair<double, double> valWithError() {
  double val = 1.1;
  double err = 0.1;
  return {val, err};
}
tuple<string, double, char> tupleReturner() {
  string name = "Hello!";
  double pi = 3.1415;
  char symbol = 'M';
  return tie(name, pi, symbol);
}
```

- `std::pair` позволяет работать с парой гетерогенных объектов
- `std::tuple` – гетерогенная коллекция фиксированного размера

```cpp
int main() {
  auto [val, err] = valWithError();
  auto [name, pi, symbol] = tupleReturner();
}
```

Пространства имён

- *C++* позволяет объединять любые элементы программы в пространства имен, `namespace`.
- Пространства имён нужны для решения проблемы пересечения имен функций или классов
- Пространства имен могут быть вложенными
- Стандартные библиотечные функции и классы *C++* определены в пространстве имен `std`

```
#include <iostream>
using namespace std;

namespace John {
  void print() {cout << "John" << endl;}
}

namespace Peter {
  void print() {cout << "Peter" << endl;}
}

int main() {
    // print(); // Ошибка!
    using Peter::print;
    print(); // Peter
    John::print(); // John
}
```

Обработка ошибок

- При выполнении программы что-то может пойти «не так», и должен быть механизм сообщения о возникновении ошибки
- В языке *C* для этого используются код возврата из функции, или глобальная переменная
- Но как, например, сообщить, что возникла ошибка в конструкторе – у него ведь нет возвращаемого значения?
- В *C++* встроен универсальный механизм сообщений о возникновении ошибок – обработка исключений. Похожий механизм `try-catch` используется в большинстве современных языков программирования

```
int divide(int i, int j) {
  if (j == 0) throw 99;
  return i / j;
}
```

```
#include <iostream>
using namespace std;
int main() {
  try {
    divide(2, 0);
  } catch (int ierr) {
    cerr << "Error! Code=" << ierr << endl;
  } catch (...) {
    cerr << "Another error" << endl;
  }
}
```

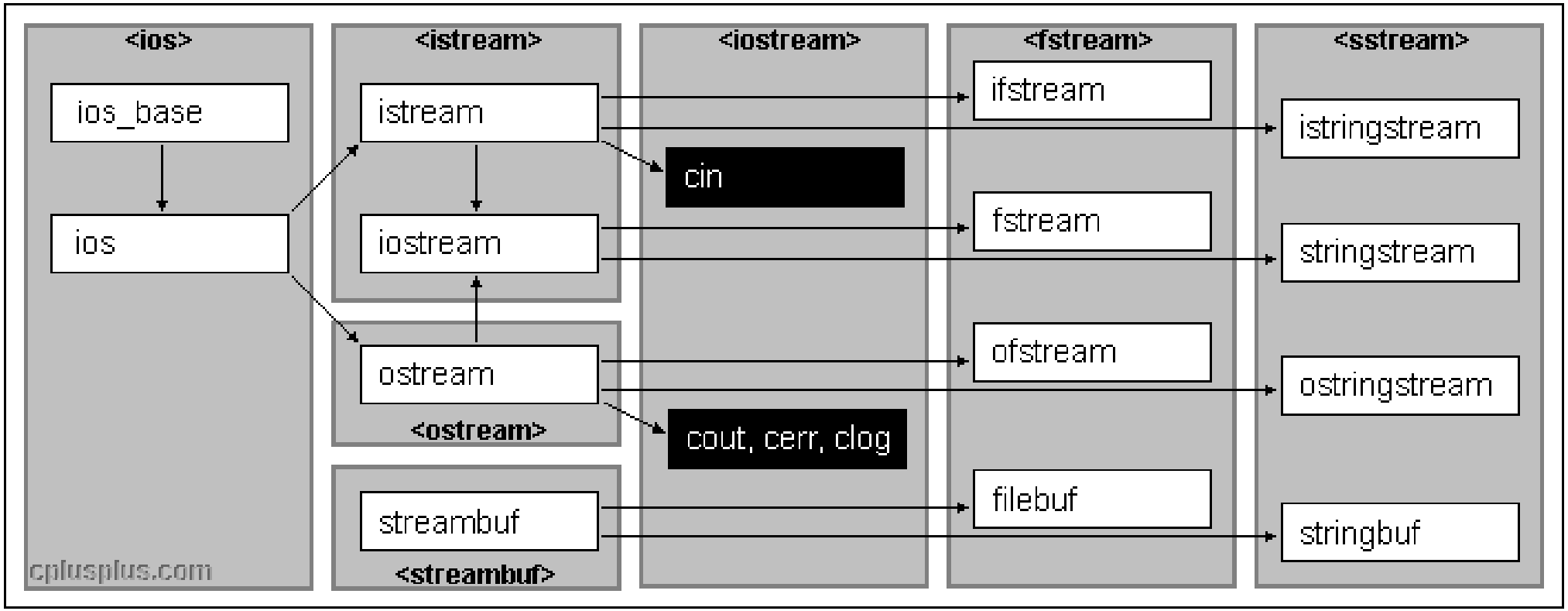
Обработка ошибок

```
#include <stdexcept>
class Time {
  int h, m, s;
public:
  Time (int hh, int mm, int ss) : h(hh), m(mm), s(ss) {
    if (mm > 59) throw std::invalid_argument("minutes > 59");
    if (ss > 59) throw std::invalid_argument("seconds > 59");
  }
};
```

```
try {
  Time now(13,61,00);
} catch (const invalid_argument& err) {
  cerr << "Error in creating now: " << err.what() << endl;
} catch (const exception& err) {
  cerr << "Something wrong: " << err.what() << endl;
}
```

Error in creating now: minutes > 59

- В заголовочном файле `<stdexcept>` определены часто встречающиеся исключения
- Классы стандартных исключений являются наследниками `std::exception`



Потоки ввода-вывода

Потоки ввода-вывода

- В *C++* для работы с любыми устройствами ввода-вывода придумана абстракция *поток* (stream). Поток можно открыть и закрыть, в него можно писать или из него можно читать
  - Оператор << добавляет данные в поток вывода (ostream)
  - Оператор >> читает данные из потока ввода (istream)
- Типы потоков:
  - Стандартные потоки `<iostream>`. Глобальные объекты:
    - `cout` – терминал
    - `cin` – клавиатура
    - `cerr` – сообщения об ошибках
  - Файловые потоки `<fstream>`
  - Строковые потоки `<sstream>`

Строковые потоки

```
#include <sstream>
#include <string>
#include <iostream>
using namespace std;
class Man {
  string name;
  unsigned int age;
  char gender;
  bool status;
public:
  friend istream& operator>>(istream& is, Man& m) {
    return is >> m.name >> m.age >> m.gender >> m.status;
  }
  friend ostream& operator<<(ostream& os, const Man& m) {
    return os << "Man: " << m.name << " " << m.age
       << " " << m.gender << " " << m.status;
  }
};
```

```
#include "man.h"

using namespace std;

int main() {
  string info("Vitaly 31 M 1");
  Man lec;
  istringstream iss(info);
  iss >> lec;
  ostringstream oss;
  oss << lec;
  cout << oss.str() << endl;
}
```

```
Man: Vitaly 31 M 1
```

Файловые потоки

- Задача: прочитать из файла массив целых чисел и вычислить их сумму

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    string fname("data\\vec.txt");
    ifstream ifile(fname, ifstream::in);
    if (!ifile.good()) {
        cerr << "Can't open file " << fname << endl;
        return 1;
    }
    int sum = 0;
    while (!ifile.eof()) {
        int n;
        ifile >> n;
        sum += n;
    }
    cout << "Sum: " << sum << endl;
}
```

Настройка потока вывода

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<double> matrix = {
    1, 100, 1, -100,
   -10.4534, 100, 1, -100,
   -158.9, 999,  0.008,  0
  };
  for (size_t row = 0; row < 3; ++row) {
    for (size_t col = 0; col < 4; ++col) {
      cout << matrix[row*4 + col] << " ";
    }
    cout << endl;
  }
}
```

- Задача: вывести к консоль матрицу 3x4
- Попытка 1

```
1 100 1 -100
-10.4534 100 1 -100
-158.9 999 0.008 0
```

Настройка потока вывода

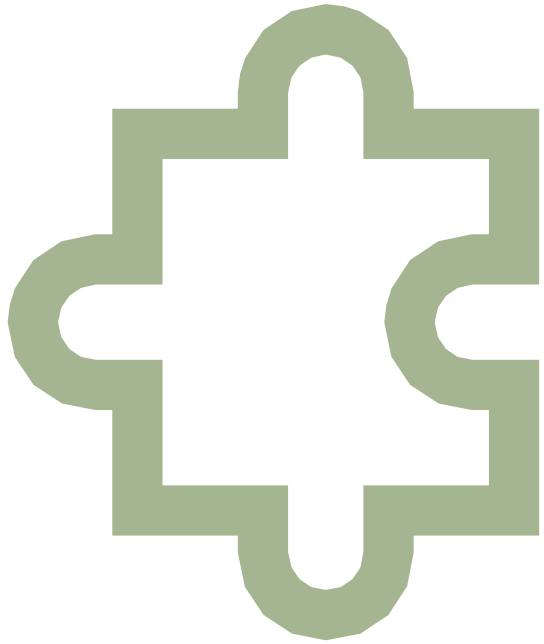
```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
  vector<double> matrix = {
    1, 100, 1, -100,
    -10.4534, 100, 1, -100,
    -158.9, 999,  0.008,  0};
  for (size_t row = 0; row < 3; ++row) {
    for (size_t col = 0; col < 4; ++col) {
      cout << setw(8) << setprecision(2) << fixed
           << matrix[row*4 + col] << " ";
    }
    cout << endl;
  }
}
```

- Задача: вывести к консоль матрицу 3x4
- Попытка 2
- Смотрите документацию <iomanip>

```
   1.00  100.00    1.00 -100.00
 -10.45  100.00    1.00 -100.00
-158.90  999.00    0.01    0.00
```

Суммируем I

1. Класс `std::string` предоставляет удобные инструменты для работы со строками
2. Классы `std::pair` и `std::tuple` полезны для возвращения из функции нескольких объектов
3. Ввод-вывод в *C++* производится через потоки с помощью операторов `<<` и `>>`
4. Глобальные переменные `cout`, `cin` и `cerr` для работы со стандартными потоками определены в файле `<iostream>`
5. Чтобы объекты класса можно было выводить в поток или читать из потока, необходимо перегрузить операторы `<<` и `>>`
6. *C++* содержит механизм обработки исключений. При возникновении ошибки генерируется (`throw`) исключение, которое можно обработать в блоке `try-catch`
7. Пространства имён являются полезным инструментом для организации кода больших проектов на *C++*



Обобщенное программирование

Обобщенное программирование

- Часто алгоритм действий не зависит от типа данных, с которыми эти действия производят. Например, алгоритм сортировки не зависит от типа данных массива
- Было бы удобно иметь универсальную функцию `swap()`, в которой описать алгоритм обмена, и научить компилятор применять ее для любых типов. Такой подход (парадигма) называется *обобщенным (generic) программированием*
- В *C++* обобщенное программирование реализуется с помощью *шаблонов*

```
void swap(int &x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

```
void swap(double &x, double & y) {
  double tmp = x;
  x = y;
  y = tmp;
}
```

Шаблоны функций

- В заголовке шаблона указывается абстрактный тип данных, для которого написана функция (алгоритм). При использовании шаблона к имени функции добавляется имя конкретного типа, для которого используется шаблон

```
template<typename T>
void swap(T& x, T& y) {
  T tmp = x;
  x = y;
  y = tmp;
}
```

```
#include <string>
int main() {
  int i=5, j=10;
  swap<int>(i,j); // i=10, j=5

  double a=5.0, b=10.0;
  swap<double>(a,b); // a=10.0, b=5.0

  std::string str = "Hello";
  swap<char>(str[1],str[2]); // "Hlelo"
}
```

Шаблоны функций

- В заголовке шаблона указывается абстрактный тип данных, для которого написана функция (алгоритм). При использовании шаблона к имени функции добавляется имя конкретного типа, для которого используется шаблон
- Шаблонную специализацию можно не указывать, если у компилятора достаточно информации

```
template<typename T>
void swap(T& x, T& y) {
  T tmp = x;
  x = y;
  y = tmp;
}
```

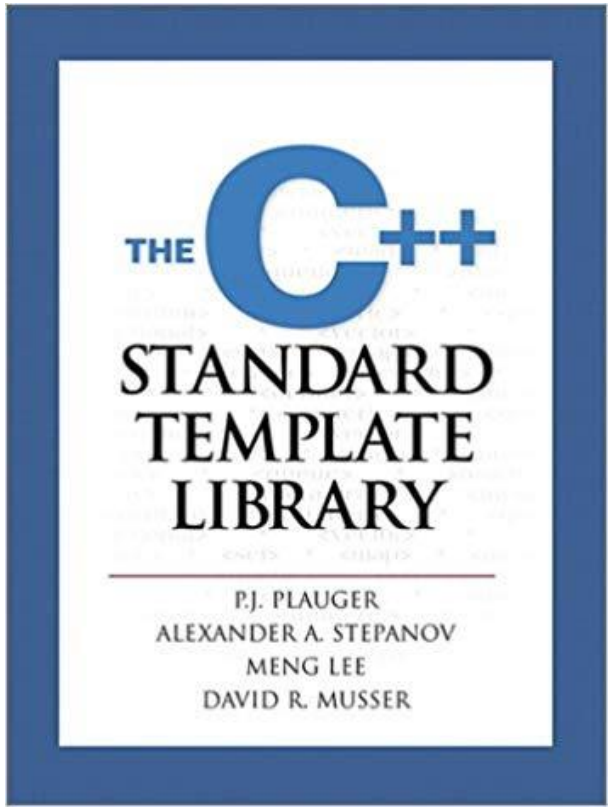
```
#include <string>
int main() {
  int i=5, j=10;
  swap(i,j); // i=10, j=5

  double a=5.0, b=10.0;
  swap(a,b); // a=10.0, b=5.0

  std::string str = "Hello";
  swap(str[1],str[2]); // "Hlelo"
}
```

Шаблоны классов

```cpp
#include <string>

template<typename T>
class KeyValue {
  std::string key;
  T value;
public:
  KeyValue(const std::string& name, const T& val) :
    key(name), value(val) {}

  const std::string& getName() const {
    return key;
  }
  const T& getValue() const {
    return value;
  }
};
```

- Задача: разработать инструмент для хранения конфигурации устройства. Каждый параметр конфигурации – это пара имя-значение
- Значения разных параметров имеют разные типы

```cpp
#include <iostream>
struct Date {int d, m, y;};
using namespace std;
int main() {
  KeyValue<int> volt("Voltage", 220);
  KeyValue<string> name("Name", "Lab1");
  KeyValue<Date> d("Installed", Date{25, 9, 2017});
}
```

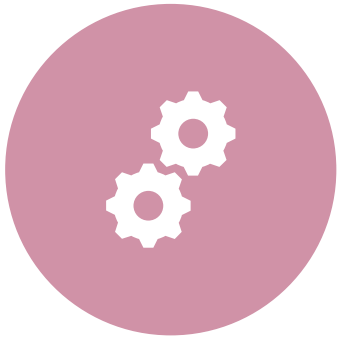


Стандартная библиотека шаблонов

STL

КОНТЕЙНЕРЫ

ИТЕРАТОРЫ



АЛГОРИТМЫ

STL контейнеры

Последовательные

- `<array>`
- `<list>`
- `<forward_list>`
- `<vector>`
- `<deque>`

Ассоциативные

- `<set>`
- `<unordered_set>`
- `<map>`
- `<unordered_map>`

Адаптеры

- `<queue>, priority_queue -> list`
- `<stack> -> vector`

- Контейнеры имеют максимально схожий интерфейс
  - По любому контейнеру можно итерироваться
  - size() возвращает количество элементов в контейнере
  - …
- Ассоциативные контейнеры предоставляют подступ к элементам по ключу

std::vector

- Хранит объекты последовательно в памяти, что обеспечивает максимально быстрое итерирование
- Предоставляет произвольный доступ к элементам (operator[](size_t))
- Вставка (push_back) и удаление (pop_back) в конец вектора за константное время
- Вставка (insert) и удаление (erase) элемента в произвольном месте за линейное время
- Поиск элемента за линейное время

```
#include <vector>
using namespace std;

int main() {
  vector<int> a = {1,2,3,4};
  vector<double> b(3);
  vector<int> c;
  for (int i = 0; i < 10; ++i)
    c.push_back(i);
  // 0 1 2 3 4 5 6 7 8 9
  for (int i : c) {
    cout << i << " ";
  }
  cout << c.front() << "  "
       << c.back();  // 0 9
}
```

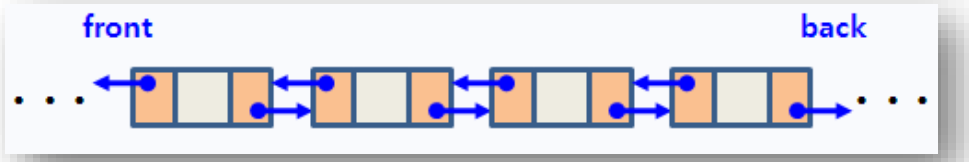
std::list

- `list` – двусвязный список

- Нет произвольного доступа к элементам. Только последовательный

- Итерирование по элементам медленнее, чем у `vector`

- Вставка и удаление элемента в произвольном месте за константное время




```cpp
#include <list>
using namespace std;

int main() {
  list<int> a = {1,2,4};
  auto it = next(next(a.begin()));
  for (int i = 5; i < 7; ++i)
    a.push_back(i);
  a.insert(it, 3);
  // 1 2 3 4 5 6
  for (int i : a)
    cout << i << " ";
}
```

std::set

- Хранит **упорядоченный** набор уникальных элементов
- Элементы в `set` должны быть comparable
- Вставка, удаление и поиск элементов за логарифмическое время

```cpp
#include <set>
#include <string>
#include <iostream>
using namespace std;

int main() {
  set<string> s {"MSU", "MIPT",
    "HSE", "MEPhI", "MIT"};
  s.insert("NSU");
  s.erase("MIT");
  auto it = s.find("NSU");
  if (it != s.end()) {
    cout << "NSU is in the list!"
         << endl;
  } else {
    cout << "NSU is NOT in the list!"
         << endl;
  }
}
```

std::map

- Хранит набор пар (key, value), упорядоченных по значению key
- Объекты key должны быть comparable
- Вставка, удаление и поиск элементов за логарифмическое время

```
France: Paris
Germany: Berlin
Italy: Rome
Russia: Moscow
Switzerland: Geneva
```

```cpp
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  map<string, string> capitals {
    {"Germany", "Berlin"},
    {"France", "Paris"},
    {"Switzerland", "Geneva"},
    {"Italy", "Rome"},
    {"Iran", "Tehran"}
  };
  capitals.insert({"Russia", "Moscow"});
  capitals.erase("Iran");
  for (auto& [country, city] : capitals) {
    cout << country << ": " << city << endl;
  }
}
```

Простая схема

Какой контейнер использовать?

STL алгоритмы

В STL реализовано множество алгоритмов работы с коллекциями:

- Перебор, поиск и изменения элементов

  count, count_if, find, find_if, for_each, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, reverse, random_shuffle, …

- Сортировка, работа с отсортированными коллекциями

  sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, min, max, min_element, max_element, …

- Выполнение арифметических операций с элементами

  accumulate, inner_product, partial_sum, …

Алгоритмы: пример

```cpp
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> v {1, 5, 4, 6, 1, 4};
  auto it = find(v.begin(), v.end(), 5);
  if (it != v.end()) cout << "Found " << *it << endl;
  // Посчитать, сколько раз встречается 1
  int n = count(v.begin(), v.end(), 1);
  cout << "Found " << n << " times" << endl;
  // Отсортировать первые 4 элемента массива
  sort(v.begin(), v.begin() + 4);
}
```

Управление сравнением I

```cpp
#include <iostream>
using namespace std;

struct Point2D {
  double x, y;
  double norm2() const { return x*x+y*y;}
};

bool operator<(const Point2D& lhs, const Point2D& rhs) {
  return lhs.norm2() < rhs.norm2();
}

ostream& operator<<(ostream& os, const Point2D& p) {
  return os << "(" << p.x << ", " << p.y << ")";
}
```

```cpp
#include <algorithm>
#include <vector>

int main() {
  vector<Point2D> vec{
    {1, 2}, {0.1, 0.3}, {-8, 1}
  };

  sort(begin(vec), end(vec));
  for (auto& p : vec) cout << p << endl;
}
```

```
(0.1, 0.3)
(1, 2)
(-8, 1)
```

- Не решение для стандартных типов
- Не решение, если мы хотим несколько разных вариантов сравнения

Управление сравнением II

```cpp
#include <iostream>
using namespace std;

struct Point2D {
  double x, y;
  double norm2() const { return x*x+y*y;}
};

bool PointCmp(const Point2D& lhs, const Point2D& rhs) {
  return lhs.norm2() < rhs.norm2();
}

ostream& operator<<(ostream& os, const Point2D& p) {
  return os << "(" << p.x << ", " << p.y << ")";
}
```

```cpp
#include <algorithm>
#include <vector>

int main() {
  vector<Point2D> vec{
    {1, 2}, {0.1, 0.3}, {-8, 1}
  };

  sort(begin(vec), end(vec), PointCmp);
  for (auto& p : vec) cout << p << endl;
}
```

```
(0.1, 0.3)
(1, 2)
(-8, 1)
```

- Слишком много букв, если нужно что-то простое

Управление сравнением III

```
#include <iostream>
using namespace std;

struct Point2D {
  double x, y;
  double norm2() const { return x*x+y*y;}
};

class PointCmp {
  bool asc;
public:
  PointCmp(bool asc=true) : asc(asc) {}
  bool operator()(const Point2D& lhs, const Point2D& rhs)
  {
    return asc ? lhs.norm2() < rhs.norm2() :
                 lhs.norm2() > rhs.norm2();
  }
};
```

```
#include <algorithm>
#include <vector>

int main() {
  vector<Point2D> vec{
    {1, 2}, {0.1, 0.3}, {-8, 1}
  };

  sort(begin(vec), end(vec), PointCmp(false));
  for (auto& p : vec) cout << p << endl;
}
```

```
(-8, 1)
(1, 2)
(0.1, 0.3)
```

Управление сравнением IV

```
#include <iostream>
using namespace std;

struct Point2D {
  double x, y;
  double norm2() const { return x*x+y*y;}
};
```

```
(0.1, 0.3)
(1, 2)
(-8, 1)
```

- Лямбда-выражения идеально подходят, когда нужна простая функция, которая будет использоваться один раз

```
#include <algorithm>
#include <vector>

int main() {
  vector<Point2D> vec {
    {1, 2}, {0.1, 0.3}, {-8, 1}
  };

  sort(begin(vec), end(vec),
    [](const Point2D& lhs, const Point2D& rhs) {
      return lhs.norm2() < rhs.norm2();
    });

  for (auto& p : vec) cout << p << endl;
}
```

Пример

- Задача. Удалить из вектора все числа кратные 3

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>

using namespace std;
int main() {
  vector<int> vec(100);
  iota(vec.begin(), vec.end(), 0); // 0, 1, 2, …
  auto it = stable_partition(vec.begin(), vec.end(),
                [](int val){return val % 3;});
  vec.erase(it, vec.end());
  for (auto v : vec) cout << v << " ";
}
```

Итераторы

```cpp
#include <vector>

using namespace std;
int main() {
  vector<int> v{1, 2, 3};
  vector<int>::iterator it = v.begin();
  vector<int>::const_iterator cit = v.cbegin();
  vector<int>::reverse_iterator rit = v.rbegin();
  vector<int>::const_reverse_iterator crit = v.crbegin();
}
```

- Да, можно не указывать каждый раз полный тип

Итераторы

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> v{1, 2, 3};
  auto it = v.begin();
  auto cit = v.cbegin();
  auto rit = v.rbegin();
  auto crit = v.crbegin();

  for (auto el = v.crbegin(); el != v.crend(); ++el) {
    cout << *el << " ";
  }
}
```

Действия с итераторами

```
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int> v{1, 2, 3, 4};
  auto it1 = v.begin();  // 1
  ++it1;  // 2
  --it1;  // 1
  auto it2 = find(begin(v), end(v), 3);  // 3
  bool is_equal = it1 == it2;  // false
  auto it3 = next(it1);  // 2
  auto it4 = prev(it3);  // 1
  auto it5 = it1 + 3;  // 4
  advance(it1, 3);  // 4
  int dist1 = it4 - it3;  // 1
  int dist2 = distance(it3, it4);  // 1
}
```

- Итераторы бывают разные:
  - random_access_iterator
  - bidirectional_iterator
  - forward_iterator
- operator-- не определён для forward_iterator

Итераторы в действии

```cpp
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <numeric>
#include <iterator>
using namespace std;

int main() {
  string fname("input.txt");
  ifstream ifile(fname, ifstream::in);
  istream_iterator<int> start(ifile), end;
  vector<int> vec(start, end);
  sort(vec.begin(), vec.end());
  int sum = accumulate(vec.begin(), vec.end(), 0);
  copy(vec.cbegin(), vec.cend(), ostream_iterator<int>(cout, " "));
}
```

- 6 строк кода
  - Открываем файл
  - Считываем массив
  - Сортируем вектор
  - Вычисляем сумму
  - Выводим в консоль

Что осталось за кадром?

- Многопоточность
- Регулярные выражения
- Генераторы случайных чисел
- …
- … …

Старайтесь узнавать больше возможностей языка и стандартной библиотеки C++. Это позволит вам писать более выразительный и эффективный код

Суммируем

1. Обобщенное программирование является самодостаточной парадигмой программирования, как и объектно-ориентированное
2. C++ поддерживает парадигму обобщенного программирования посредством шаблонов (templates)
3. Стандартная библиотека C++ имеет реализацию основных структур данных и достаточно большой набор алгоритмов для работы с этими структурами
4. Итераторы инкапсулируют логику перемещения по элементам контейнера и позволяют единообразно использовать разные контейнеры