Обзор STL. Часть 1

Введение объектно-ориентированное программирование

13.03.2025

Standard input/output stream

```
int a, b;
std::cin » a » b; //operator »
std::cout « “x=“ « x « “ , y=“ « y « std::endl;
```

Файловый поток

Пример построчного чтения файла

```
std::ifsteam inp(“text.txt“);
if (!inp) {
   std::cerr « “File not found!“;
   return 1;
}
std::string s;
for (int n = 1; getline(inp, s); s++)    std::cout « “line“ « n « “: “ « s « std::endl;
```

Запись в файл и stringstream

```
std::string getline(int n) {
  std::ostringstream oss;
  oss « n « “n square = “ « n*n;
  return oss.str(); }
std::ofstream of;
of.open(“out.txt“, std::ofstream::out);
if (!of) {
  std::cerr « “cant open file“;
  return 1;
}
for (int i = 0; i < 10; ++i) {
  std::string line = getline(i);
  of « line « std::endl;
}
of.close();
```

Работа со строками

```
std::string s(“abcd“);
s.size(); // 4
s.clear() // чистим строку
s.empty() // пустая ли строка
s += “1234“;
s[3] // 4
s.insert(1, “abc“); //1abc234
s.erase(2, 2); //1a234
s.replace(0, 2, A1); //a1234
s.substr(2); //234
s.substr(2, 1) // 2
```

Работа по указателю и поиск

```
std::string s(“abcdab“);
s.c_str(); // const char*
size_t pos = s.find(“ab“); // 0
size_t rpos = s.rfind(“ab“); // 4
auto what = s.find(“wat“);
if (what == std::string::npos)
   std::cout « “not found“ « std::endl;
```

Разбиение строки на слова

```
std::string s;
std::getline(std::cin, s);
for (size_t pos =0;;) {
   pos = s.find_first_of(“\t“, pos);
   if (pos == std::string::npos)
     break;
  auto pos1 = s.find_first_of(“\t“, pos);
  auto len = (pos1 == std::string::npos) ? std::string::npos : pos1 - pos;
  std::string word(s.substr(pos, len));
  if (pos1 == std::string::npos)
     break;
  pos = pos1;
}
```

Динамический массив

```
std::vector<int> v;
v.push_back(10);
v.push_back(10);
v.push_back(10);
v[0]; // 10;
std::vector<int> zeroes(100), fives(10, 5);
v.front(); // 10
```

Итераторы: отсутпление

Итератор можно воспринимать как класс, который ведет себя как указатель на элемент в контейнере. Будь то указатель в массиве, уазатель в списке или указатель на элемент дерева.

Итераторы

1. Обращение к элементу // input output iterator
2. Сдвиг вперед на I // forward iterator
3. Сдвиг назад на I // backward iterator
4. Сдвиг вперед на N // random access iterator
5. Сдвиг назад на N // random access iterator
6. Разность между двумя итераторами

Итераторы у std::vector

```
std::vector<int> a = {1, 2, 3, 4};
for (std::vector<int>::const_iterator it = a.begin(); it != a.end(); ++it)
  std::cout « *it;
```

auto и range-based for-loop (C++11)

```
std::vector<int> a = {1, 2, 3, 4};
for (auto it = a.begin(); it != a.end(); ++it)
  std::cout « *it;
for (auto &x: a)
  ++x;
for (auto x: a)
  std::cout « x « ‘ ‘; // 2 3 4 5
```

Обратные итераторы

```
std::vector<int> a = {1, 2, 3, 4};
for (auto it = a.begin(); it != a.end(); ++it)
   std::cout « *it; // 4 3 2 1
```

Операции с std::vector

```
std::vector<int> a = {1, 2, 3, 4, 5};
a.resize(6); // 1 2 3 4 5 0
a.resize(8, 99); // 1 2 3 4 5 0 99 99
a.resize(3); // 1 2 3
a.pop_back() // 1 2
a.insert(a.begin() + 1, 7) // 1 7 2
std::vector<int> b = {-1, -2 };
a.insert(a.begin(), b.begin(), b.end()); // -1 -2 1 7 2
a.erase(a.begin() + 2); // -1 -2 7 2
a.swap(b); // a<==> b
a.clear()
```

Последовательные контейнеры

- Контейнеры
  - vector — массив с переменным размером и гарантией непрерывности памяти*
  - array — массив с фиксированным размером, известным в момент компиляции
  - deque — массив с переменным размером без гарантий по памяти
  - list — двусвязный список
  - forward_list — односвязный список
- Адаптеры
  - stack — LIFO контейнер, чаще всего на базе deque
  - queue — FIFO контейнер, чаще всего на базе deque
  - priority_queue — очередь с приоритетами, чаще всего на базе vector

Что может смущать в этом коде?

```
std::deque<int> d; // подумайте если бы это был vector?
for (int i = 0; i != N; ++i) {
d.push_front(i);
d.push_back(i);
}
```

- deque — массив с переменным размером без гарантий по памяти
- Поэтому ответ: всё хорошо.
- Вставка в начало и в конец дека имеет всегда честную константную сложность O(1).

Fill Bench

Рассмотрите deque вместо vector*

- Эффективно растёт в обоих направлениях
- Не требует больших реаллокаций с перемещениями, так как разбит на блоки
- Гораздо меньше фрагментирует кучу

Деки против векторов

Вектора

- Доступ к элементу O(1)
- Вставка в конец аморт. O(1)+
- Вставка в начало O(N)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Есть гарантии по памяти
- Есть reserve / capacity

Замеры

Деки против векторов

Деки

- Доступ к элементу O(1)
- Вставка в конец O(1)
- Вставка в начало O(1)
- Вставка в середину O(N)
- Вычисление размера O(1)
- Нет гарантий по памяти
- Нет необходимости в reserve/capacity

Узловые контейнеры

- **deque** произвольный доступ, быстрая вставка в начало и в конец.
- **forward_list** последовательный доступ, быстрая вставка в любое место.
- **list** последовательный доступ, быстрая вставка в любое место, итерация в обе стороны.

Сплайс для списков: простая форма

```
// forward_list<int> fst = {10, 20, 30, 1, 2, 3 };
// forward_list<int> snd = {};
// it указывает на 1
// перекидываем элементы со второго по it в список second
snd.splice_after(snd.before_begin(), fst, fst.begin(), it);
```

Сплайс для списков: сложная форма

```
// forward_list<int> fst = {10, 20, 30, 1, 2, 3 };
// forward_list<int> snd = {};
// it указывает на 1
// перекидываем элементы со второго по it в список second
snd.splice_after(snd.before_begin(), fst, fst.begin(), it);
```

Сплайс для списков: средняя форма

```
// forward_list<int> fst = { 10, 1, 2, 3 };
// forward_list<int> snd = { 20, 30 };
// it указывает на 1
// все элементы второго списка начиная со второго в первый
fst.splice_after(fst.before_begin(), snd, snd.begin());
```

Виды адаптеров

• **stack** – LIFO стек над последовательным контейнером
template <class T, class Container = deque<T> > class stack;
• **queue** – FIFO очередь над последовательным контейнером
template <class T, class Container = deque<T> > class queue;
• **priority_queue** – очередь с приоритетами (как binary heap) над последовательным контейнером
template <class T, class Container = vector<T>, class Compare = less<typename Container::value_type» class priority_queue;

Защита от ортогональности

std::stack<int> s; // ok, это stack <int, deque<int»
std::stack<int, std::vector<long» s1; // сомнительно
std::stack<int, std::vector<char» s2; // совсем плохо
s2.push(1000);
// Что вернёт s2.top()?

- К счастью всё это безобразие перекрыто static asserts

Недостаточная ортогональность

std::stack<int, std::forward_list<int» s; // ok
s.push(100); // ошибка: нет push_back
s.pop(); // ошибка: нет pop_back
s.top(); // ошибка: нет back

- Эти ошибки неочевидны
- Стек вполне может быть сделан на односвязном списке
- Но адаптер std::stack требует (неявно требует) вполне определённый интерфейс

Проблема статических строк

• Что вы думаете об использовании константных статических строк?

```
static const std::string kName = “oh literal, my literal“;
// .....
int foo(const std::string &arg);
// .....
foo(kName);
```

Решение: string_view (C++17)

- string_view это невладеющий указатель на строку

```
static std::string_view kName = “oh literal, my literal“;
// .....
int foo(std::string_view arg);
// .....
foo(kName);
```

- Здесь нет ни heap indirection ни создания временного объекта

Базовые операции над string_view

- remove_prefix
- remove_suffix
- copy
- substr
- compare
- find
- data

Views: идея для span (C++20)

- std::span для одномерных массивов то же, что string_view для строк

```
int arr[4] = {1, 2, 3, 4}; // просто данные
std::array<int, 4> arr = {1, 2, 3, 4}; // копирование до main
```

- span решает эту проблему

```
std::span<int, 4> arr = {1, 2, 3, 4}; // просто данные
```

- По умолчанию второй параметр N это std::dynamic_extent

```
std::span<int> dynarr(arr); // неизвестный размер
```

- Разумеется у него куда более простой интерфейс, чем у string view.

Обсуждение

- Хватит ли нам последовательных контейнеров?