Основы разработки на C++: красный пояс

Неделя 4

Эффективное использование линейных контейнеров

Оглавление

Эффективное использование линейных контейнеров

4.1. Эффективное использование линейных контейнеров

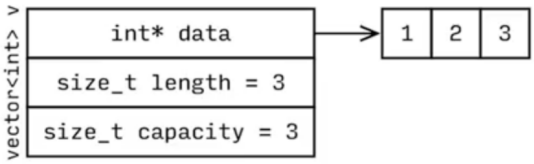
Мы приступаем к исследованию последовательных или линейных контейнеров. Так называют контейнеры, которые сохраняют порядок вставляемых в них элементов. Типичный пример – вектор. Пример непоследовательного контейнера – множество.

4.1.1. Эффективное использование вектора

Создадим вектор из трёх чисел:

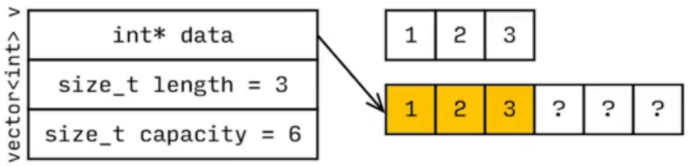
```
vector<int> v = {1, 2, 3};
```

На стеке у этого вектора будет указатель на кучу, также на стеке будет храниться длина этого вектора, а также `capacity` – количество памяти, доступное данному вектору в куче.



Что будет, если мы захотим добавить четвёрку в вектор? Нам нужен блок памяти, в который мы сможем положить `1`, `2`, `3` и `4`. Соответственно, как и в задаче `SimpleVector`, стандартный вектор выделяет в два раза больше памяти, чем у него было до этого.

Он выделяет блок под 6 `int`'ов, чтобы туда можно было положить четыре числа. Теперь вектор присваивает себе блок памяти из шести элементов, про старый забывает, в новый копирует



три `int`'а, которые у него были, и теперь туда же может положить четверку, предварительно освободив старый блок памяти. Теперь длина – 4, `capacity` – 6.

Напишем функцию `LogVectorParams`, которая будет для вектора выводить его параметры: длину и его `capacity`.

```
void LogVectorParams (const vector<int>& v) {
  cout << "Length = " << v.size() << ", " <<
      "capacity = " << v.capacity() << "\n";
}
```

Вызовем функцию после создания вектора и после добавления четвёрки.

```
int main() {
  vector<int> v = {1, 2, 3};
  LogVectorParams(v);
  v.push_back(4);
  LogVectorParams(v);

  return 0;
}
// Length = 3, capacity = 3
// Length = 4, capacity = 6
```

Всё, как мы указали. Мы утверждали, что элементы вектора хранятся подряд. Более того, в `SimpleVector`'е мы оперировали указателями и там был указатель на данные в куче. Оказывается, стандартный вектор тоже может отдать нам этот указатель.

У стандартного вектора есть метод `data`, который возвращает указатель на те данные, которые у него лежат в куче. Вызовем этот метод, сохраним его в указателе. Мы хотим посмотреть на данные, которые лежат по указателю `data` и в следующих ячейках. Вектор константный, поэтому `data` возвращает константный указатель, поэтому переменная `data` должна иметь тип `const int*`.

```
int* data = v.data();
```

```
for (size_t i = 0; i < v.capacity(); ++i) {
  cout << *(data + i) << " ";
}
cout << "\n";

// Length = 3, capacity = 3
// 1 2 3
// Length = 6, capacity = 6
// 1 2 3 4 78063648 0
```

С помощью указателей мы посмотрели на содержимое вектора: там лежат 1, 2, 3, 4 и два числа, которые когда-то лежали в куче. Вектор их никак не инициализировал, в эту память вектор потом сможет добавить новые числа.

Если мы захотим очистить память от этих двух чисел, то можем использовать `shrink_to_fit()`. Добавим его в программу и посмотрим, что получится.

```
v.shrink_to_fit();
LogVectorParams(v);
// Length = 4, capacity = 4
// 1 2 3 4
```

Рассмотрим пример. Допустим, мы хотим сложить в вектор какие-то числа, заранее зная, сколько этих чисел будет. Подадим на вход число 10000000.

```
int main() {
  int size;
  cin >> size;

  { LOG_DURATION("push_back");
    vector<int> v;
    for (int i = 0; i < size; ++i) {
      v.push_back(i);
    }
  }

  return 0;
}
// push_back: 427 ms
```

Казалось бы, за секунды мы должны успевать больше операций. Такая серия `push_back`'ов неэф-

фективна за счёт того, что при `push_back`'е вектор при необходимости перевыделяет память и копирует данные из старой памяти в новую. В данном случае мы знаем, что вектор будет иметь размер `size`. Пусть вектор заранее выделит себе блок из `size` элементов.

```
{ LOG_DURATION("push_back");
    vector<int> v;
    v.reserve(size);
    for (int i = 0; i < size; ++i) {
      v.push_back(i);
    }
}
// push_back: 288 ms
```

4.1.2. Инвалидация ссылок

У нас есть вектор из трех элементов, что будет, если перед добавлением четвертого элемента сохранить ссылку на первый элемент?

```
int main() {
  vector<int> v = {1, 2, 3};
  int& first = v[0];
  cout << first << "\n";
  v.push_back(4);
  cout << first << "\n";

  return 0;
}
// 1
// 45628208
```

До `push_back`'а там лежала `1`, после `push_back`'а мы знаем, что вектор переместил данные в другое место, а старые данные освободил. Теперь по ссылке лежит мусорное число. Ссылка перестала быть валидной – инвалидировалась.

В материале «Класс `StringSet`» представлена реализация контейнера, в который можно добавлять строки с заданным приоритетом с помощью метода `Add`, а также находить последнюю добавленную строку и строку с наибольшим приоритетом.

В предложенной реализации строки складываются в вектор, также они складываются в `set` с

помощью структур, которые называются `StringItem`. В каждой структуре лежит сама строка вместе с приоритетом. Также есть оператор сравнения, который сравнивает `StringItem`'ы по приоритету. При добавлении строчки в контейнер мы кладем её в вектор и в `set`. В методе `FindLast()` мы берем последний элемент вектора, в методе `FindBest()` мы берём последний элемент `set`'а.

Создадим контейнер `strings`, кладём туда строчку `"upper"` с высоким приоритетом, затем строчку `"lower"` с низким приоритетом. Метод `FindLast()` должен вывести последнюю добавленную строчку, то есть `"lower"`, а метод `FindBest()` должен вывести строчку с наивысшим приоритетом, то есть `"upper"`.

```
int main() {
  StringSet strings;
  strings.Add("upper", 10);
  strings.Add("lower", 0);
  cout << strings.FindLast() << "\n";
  cout << strings.FindBest() << "\n";
  return 0;
}
// lower
// upper
```

Является ли эффективным складывать одну и ту же строчку и в вектор и в множество? Хочется этого избежать и складывать строчки в один контейнер. Можно в один контейнер сложить строчки, а в другой складывать ссылки на строчки в том контейнере. Во множество теперь будем класть не саму строчку, а ссылку на неё. В методе `Add` в `sorted_data` вставляем теперь ссылку на строчку, которую мы добавили в вектор.

```
sorted_data.Insert(StringItem{data.back(), priority});
```

В результате работы программы на экран выведется только `lower`. `upper` не выведется, потому что произошла инвалидация.

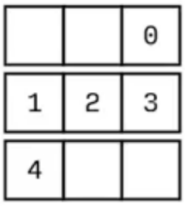
Чтобы решить проблему можно использовать контейнер, который не обладает таким недостатком, как вектор, например, `deque`.

Что делать, если важно, чтобы оставался вектор, а не `deque`? Тогда придется отказаться от ссылок. Например, вместо ссылок можно хранить индексы элементов в векторе.

4.1.3. Эффективное использование дека

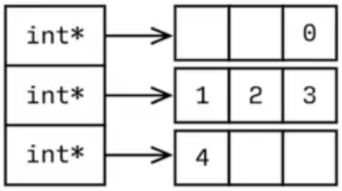
Дек можно получить, отказавшись от требования хранить все элементы подряд в едином куске памяти. Выделим кусок памяти и положим туда три `int`'а.

```
deque<int> d = {1, 2, 3};
```

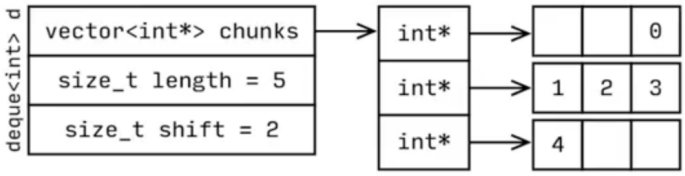


Если мы хотим добавить туда четверку, то не можем добавить её туда сразу после тройки, потому что выделили блок памяти под три `int`'а. Но мы можем выделить ещё один блок памяти на три `int`'а и положить в первую ячейку четвёрку. Аналогично, если мы хотим добавить в начало ноль, то нужно выделить блок памяти на три `int`'а и в последний элемент, чтобы соблюсти некий разумный порядок, положить ноль.

Нам нужно уметь обращаться к элементу по его номеру. Необходимо эти блоки памяти – они называются чанками – как-то проиндексировать. Сохраним вектор указателей на эти чанки, на эти блоки памяти.



В самом деке нужно хранить вектор указателей на чанки, количество элементов, также понадобится хранить так называемый сдвиг. Сдвиг того самого нуля от начала его чанка, то есть, сдвиг первого элемента дека от начала его блока памяти.



Как теперь найти второй элемент? Прибавляем сдвиг, с учетом сдвига необходим четвертый элемент. Размер блока памяти – `3`. Делим `4` на `3`, получаем `1`, идем в первый чанк памяти и там идем в элемент `4 % 3`, то есть берем остаток от деления и получаем, что в первом блоке памяти нам нужен первый элемент.

В итоге мы получили быструю вставку в начало и неинвалидацию ссылок при вставке в начало или в конец. Однако теперь мы тяжелее получаем элемент по индексу. Итерироваться тоже непросто, потому что нужно перепрыгивать между чанками.

Пусть нам нужно вставить набор чисел в наш контейнер, при этом мы заранее не знаем, сколько будет чисел – то есть не можем вызывать `reserve`.

```
int main() {
  const int SIZE = 1000000;

  vector<int> v;
  { LOG_DURATION("vector");
    for (int i = 0; i < SIZE; ++i) {
      v.push_back(i);
    }
  }

  deque<int> d;
  { LOG_DURATION("deque");
    for (int i = 0; i < SIZE; ++i) {
      d.push_back(i);
    }
  }

  return 0;
}
// vector: 61 ms
// deque: 52 ms
```

Особой разницы нет. Что, если элементов 5000000?

```
// vector: 398 ms
// deque: 168 ms
```

Это говорит о том, что нельзя заранее сказать, что будет полезнее: вектор или дек. Нужно измерять.

Теперь попробуем отсортировать полученные вектор и дек.

```
{ LOG_DURATION("sort vector");
  sort(rbegin(v), rend(v));
}
{ LOG_DURATION("sort deque");
  sort(rbegin(d), rend(d));
}
// sort vector: 5468 ms
```

```
// sort deque: 10851 ms
```

Мы быстрее заполнили дек, но сортировали его гораздо дольше, чем вектор.

Вектор хорош быстрыми обращаениями к элементам. Итерирование по вектору тоже быстрое. Дек хорош тем, что можно быстро сделать серию `push_back`'ов, если заранее не известен размер. Кроме того, можно быстро вставлять в начало, при этом не инвалидируя ссылки на элементы дека.

4.1.4. Инвалидация итераторов

Создадим вектор из одного элемента и сохраним итератор на это число. Затем вставим 2000 чисел в вектор. Посмотрим, что будет лежать по этому итератору.

```
int main() {
  vector<int> numbers = {1};
  auto it = begin(numbers);
  cout << *it << "\n";

  for (int i = 0; i < 2000; ++i) {
    numbers.push_back(i);
  }

  cout << *it << "\n";


  return 0;
}
// 1
// 78067936
```

Итераторы вектора по сути являются указателями. Соответственно при вставке в вектор они инвалидируются. Что будет для дека?

```
// 1
// 1
```

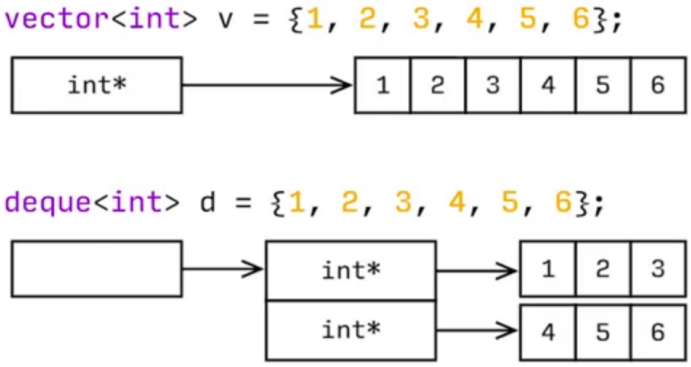
Итератор не пострадал. Попробуем с помощью этого итератора посмотреть на последний элемент дека.

```
cout << *it << " " << *(it + numbers.size() - 1) << "\n";
```

Такая программа падает. Дело в том, что итератор был получен до добавления в дек элементов. Этот итератор знает только про устройство старого дека. Итераторы содержат дополнительную логику, позволяющую итерироваться по контейнеру. Дек не может сохранять итераторы валидными.

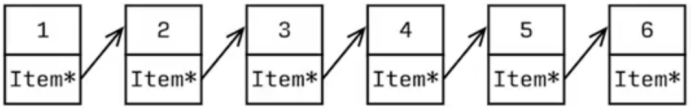
4.1.5. Контейнер `list`

Мы узнали, что есть различные стратегии выделения памяти под линейные контейнеры: вектор выделяет единый блок памяти под все свои элементы, а дек выделяет память чанками.

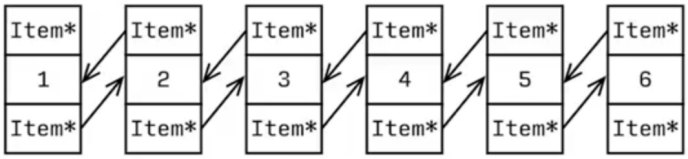


Что, если каждому элементу выделять свой кусок памяти? Такая стратегия используется в контейнере `list`.

Если каждому элементу контейнера соответствует свой блок памяти, то по такому контейнеру даже проитерироваться нельзя. Можно научить каждый элемент ходить к следующему, то есть рядом с ним положить указатель на следующий элемент.



Теперь мы можем проитерировать по списку из начала в конец. Иногда удобнее проитерироваться в обратную сторону, поэтому стоит положить указатель на обратный элемент.



Теперь мы можем быстро удалять элементы из середины списка. Если мы хотим удалить элемент 4, то мы находим его, удаляем и перевешиваем указатели.

```
list<int> numbers = {1, 2, 3, 4, 5, 6};
auto it = find(begin(numbers),
               end(numbers), 4);  // O(N)
numbers.erase(it);  // O(1)
```

Все указатели и итераторы, кроме указателя на 4, останутся валидными. Однако при таком устройстве списка мы не можем быстро обратиться к элементу по его номеру.

Рассмотрим пример. Будем складывать их в контейнер с помощью метода `Add`. Удалять элементы будем с помощью метода `Remove`, который будет принимать условие, по которому нужно удалять элементы. Он будет шаблоннным и будет принимать функциональный объект.

```
class NumbersOnVector {
public:
  void Add(int x) {
    data.push_back(x);
  };

  template <typename Predicate>
  void Remove(Predicate predicate){
    data.erase(
        remove_if(begin(data), end(data), predicate),
        end(data));
  }
private:
  vector<int> data;
};
```

Давайте реализуем этот контейнер с помощью списка.

```
class NumbersOnList {
public:
  void Add(int x) {
```

```
    data.push_back(x);
  };

  template <typename Predicate>
  void Remove(Predicate predicate){
    data.remove_if(predicate);
  }
private:
  list<int> data;
};
```

Напишем бенчмарк, чтобы сравнить два контейнера. Заполним контейнеры миллионом элементов, затем будем удалять их с помощью метода `Remove`: в начале элементы, остаток от деления на 10 у которых равен нулю, затем единице и так далее. В итоге за 10 вызовов метода `Remove` мы удалим все числа.

```
const int SIZE = 1000000;
const int REMOVAL_COUNT = 10;

int main() {
  {  LOG_DURATION("vector");
     NumbersOnVector number;
     for (int i = 0; i < SIZE; ++i) {
       numbers.Add(i);
     }
     for (int i = 0; i < REMOVAL_COUNT; ++i) {
       numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
     }
  }

  {  LOG_DURATION("list");
     NumbersOnList number;
     for (int i = 0; i < SIZE; ++i) {
       numbers.Add(i);
     }
     for (int i = 0; i < REMOVAL_COUNT; ++i) {
       numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
     }
  }
  return 0;
}
```

```
// vector: 224 ms
// list: 433 ms
```

Мы удаляли по много элементов за раз. Попробуем удалять понемногу. Пусть будет `SIZE = 10000` элементов и мы их попробуем удалить за `REMOVAL_COUNT = 1000` шагов.

```
// vector: 154 ms
// list: 109 ms
```

Теперь большую роль играет тот факт, что из списка можно быстро удалить из середины. Получается, от конфигурации входных параметров зависит, что эффективнее: вектор или список.

На том же примере продемонстрируем, что при удалении элементов из списка у нас неинвалидируются ссылки, указатели и даже итераторы. Дополним класс `NumbersOnList`, который будет находить последний элемент, удовлетворяющий некоторому условию.

```
auto FindLast(Predicate predicate) {
  return find_if(rbegin(data), rend(data), predicate);
}
```

Перед серией удалений из списка мы хотим найти последний элемент, который делится на `REMOVAL_COUNT`. Оставим в контейнере те элементы, которые делятся на `REMOVAL_COUNT`, при удалении мы будем итерироваться, начиная с единицы. В конце посмотрим на элемент, который мы нашли перед серией удалений.

```
{  LOG_DURATION("list");
   NumbersOnList number;
   for (int i = 0; i < SIZE; ++i) {
     numbers.Add(i);
   }
   auto it = numbers.FindLast(
     [](int x) { return x % REMOVAL_COUNT == 0;});
   for (int i = 1; i < REMOVAL_COUNT; ++i) {
     numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
   }
   cout << *it << "\n";
}
// list: 133 ms
// 9000
```

Итератор указывает на верный элемент. Проитерируемся по нему до начала списка.

```
while() (*it != 0) {
  cout << *it << " ";
  ++it;
}
// 9000 8000 7000 6000 5000 4000 3000 2000 1000
```

Проитерироваться получилось. Мы вывели весь список целиком с помощью итератора, который создали еще тогда, когда мы из списка не поудаляли много элементов. Это доказывает, что списки страхуют нас от инвалидации.

4.1.6. Контейнер array

Пусть нам необходимо вызывать функцию `COUNT` раз. Функция всегда возвращает пять чисел.

```
vector<int> BuildVector(int i) {
  return {i, i + 1, i + 2, i + 3, i + 4};
}

const int COUNT = 1000000;

int main() {
  { LOG_DURATION("vector");
    for (int i = 0; i < COUNT; ++i) {
      auto numbers = BuildVector()
    }
  }

  return 0;
}
// vector: 260 ms
```

Программа будет работать быстрее, если каждый раз выделять память не в куче, а в стеке. Будем использовать кортеж, он хранит свои данные на стеке.

```
tuple<int, int, int, int, int> BuildTuple(int i) {
  return make_tuple(i, i + 1, i + 2, i + 3, i + 4);
}

int main() {
```

```
  { LOG_DURATION("tuple");
    for (int i = 0; i < COUNT; ++i) {
      auto numbers = BuildTuple()
    }
  }
  return 0;
}
// tuple: 118 ms
```

Программа работает быстрее, но использование `tuple` влечет неудобства: по нему нельзя нормально проитерироваться, нельзя обратиться к элементу с использованием квадратных скобок. Необходим аналог контейнера `vector`, но на стеке. Можем попросить выделить на стеке пять `int`'ов, используя тип `array`.

```
array<int, 5> BuildArray(int i) {
  return {i, i + 1, i + 2, i + 3, i + 4};
}
```

Фрейм функции должен занимать фиксированное количество байт, поэтому прямо в массиве нужно указать, что необходимо 5 `int`'ов. 5 в данном случае – это шаблонный параметр класса. До этого мы сталкивались с классами, у которых шаблонный параметр это тип.

```
{ LOG_DURATION("array");
  for (int i = 0; i < COUNT; ++i) {
    auto numbers = BuildArray()
  }
}
// array: 9 ms
```

Почему массив настолько эффективнее, чем кортеж? Дело в том, что мы компилируем без оптимизаций. Если компиляция происходит без оптимизации, код может быть очень не эффективен. Если скомпилировать код, используя оптимизации. то результат работы программы для разных контейнеров будет таким:

```
// vector: 127 ms
// tuple: 0 ms
// array: 0 ms
```

Между `array` и `tuple` в данном случае нет разницы.

Сложность алгоритма с использованием каждого контейнера одинакова. Разница во временах

работы программ вытекает из разницы в константе, в накладных расходах, требуемых для каждого контейнера.

Продемонстрируем, что данные массива хранятся на стеке.

```
int main() {
  int x = 111111;
  array<int, 10> numbers;
  numbers.fill(8);
  int y = 222222;

  for (int* p = &y; p <= &x; ++p) {
    cout << *p << " ";
  }
  cout << "\n";

  return 0;
}
// 222222 8 8 8 8 8 8 8 8 8 123 0 111111
```

В начале идет переменная `y`, затем идёт десять восьмёрок, затем некоторая служебная информация, и переменная `x`. Действительно, всё лежит на стеке.

4.1.7. Класс `string_view`

Класс `string` похож на `vector`: он позволяет делать `push_back` и `reserve`. Однако семантика строки иногда отличается от семантики вектора.

Рассмотрим как пример задачу из второго курса, где надо было по пробелам строку разбивать на слова.

```
vector<string> SplitIntoWords(cons string& str) {
  vector<string> result;

  auto str_begin = begin(str);
  const auto str_end = end(str);

  while (true) {
    auto it = find(str_begin, str_end, ' ');
```

```
    result.push_back(string(str_begin, it));

    if (it == str_end) {
      break;
    } else {
      str_begin = it + 1;
    }
  }

  return result;
}
```

Функция итерируется по строке, на каждом шаге ищет пробел, создает строчку от текущего итератора до этого пробела, добавляет её в наш набор строк, двигается после этого на позицию, следующую за пробелом. Получается набор строк, разделённых пробелами, то есть слов. У этого кода есть очевидный недостаток. Можно было бы просто сказать, где в этой строке находятся слова, например, с первого до третьего элемента, с пятого до седьмого и так далее. Вместо этого мы стали создавать новые строки, выделять под них память, складывать эти строки в вектор.

Эта функция может возвращать не вектор, а некоторую ссылку на диапозон символов в строке. Для этого в стандарте C++17 доступен класс `string_view` – это ссылка на где-то хранящийся диапазон символов.

Новая функция будет возвращать `string_view`, принимать строчку `s`, внутри себя будет сохранять её в `string_view`. Далее появляется проблема: `string_view` не дружит с итераторами. `string_view` ссылается на подряд идущий диапазон символов. Он работает с позициями в строках, а не с итераторами. Мы начинаем не с итератора, а с нулевой позиции.

```
size_t pos = 0;
```

Аналогом итератора, указывающего за конец, является позиция за концом, её можно проинициализировать константой `str.npos` – это большое число, по сути означает несуществующую позицию.

```
const size_t pos_end = str.npos;
```

Будем искать пробел в `string_view str`, начиная с позиции `pos`.

```
size_t space = str.fins(' ', pos);
```

Теперь нужно положить в результат подстроку текущего `string_view`, начиная от позиции `pos` и до пробела. Есть метод `substr`, который возвращает `string_view` на подстроку. Нужно начать подстроку с позиции `pos`, второй элемент – длина этой подстроки. Также нужно учесть случай, когда пробел не нашелся и `space` равно `npos`.

```
result.push_back(
  space == pos_end
  ? str.substr(pos)
  : str.substr(pos, space - pos));
```

Добавив очередное слово, нужно проверить, если пробел уже не нашелся, то все слова найдены.

```
if (space == pos_end) {
  break;
} else {
  pos = space + 1;
}
```

Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
  string_view str = s;

  vector<string_view> result;

  size_t pos = 0;
  const size_t pos_end = str.npos;

  while (true) {
    size_t space = str.find(' ', pos);
    result.push_back(
        space == pos_end
        ? str.substr(pos)
        : str.substr(pos, space - pos));


    if (space == str_end) {
      break;
    } else {
      pos = space + 1;
    }
  }
```

```
  return result;
}
```

Проверим этот код, сравнив его эффективность со старой функцией `SplitIntoWords`. Для этого заготовлена функция `GenerateText`, генерирующая текст из 10000000 букв `"a"`, разбивающая его пробелами на слова через каждые сто символов.

```
int main() {
  const string text = GenerateText;
  { LOG_DURATION("string");
    const auto words = SplitIntoWords(text);
    cout << words[0] << "\n";
  }

  { LOG_DURATION("string_view");
    const auto words = SplitIntoWordsView(text);
    cout << words[0] << "\n";
  }
}
// *слово, состоящее из ста букв "a"*
// string: 188 ms
// *слово, состоящее из ста букв "a"*
// string_view: 22 ms
```

Сделаем код более компактным. Не будем держать единый `string_view` и ходить по нему позицией, а будем двигать начало `string_view`, когда обработаем сколько-то слов. В каждый момент `string_view` будет указывать на диапазон ещё необработанных символов. Если `string_view` указывает на ещё необработанные символы, то можно просто искать первый пробел:

```
size_t space = str.find(" ");
```

Подстроку будем брать не от `pos`, а от нулевой позиции. Если хотим взять подстроку от нулевой позиции до какой-то другой, возможно несуществующей, то можно брать субстроку от нуля до `space`.

```
result.push_back(str.substr(0, space));
```

Если пробел нашелся, то `space` – это его позиция, а также расстояние до него. Если пробел не нашелся, то вызываем `substr` от нуля до `npos` и это будет вся строка.

Теперь заканчиваем цикл. Если пробел не нашелся, то есть его позиция `npos`, то надо закончить

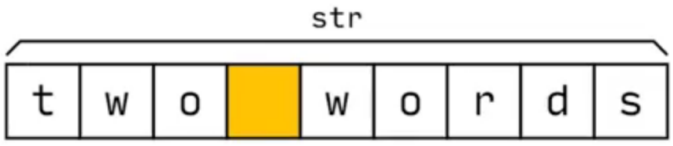
цикл, иначе надо откусить от начала `string_view` уже обработанный кусок длины `space + 1` с помощью специального метода `remove_prefix`.

```
if (space == str.npos) {
  break;
} else {
  str.remove_prefix(space + 1);
}
```

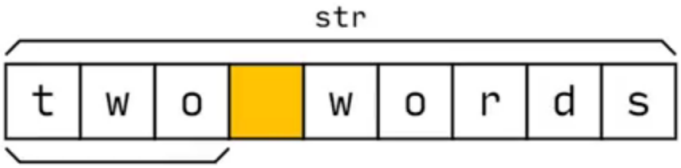
Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
  string_view str = s;

  vector<string_view> result;

  while(true) {
    size_t space = str.find(' ');
    result.push_back(str.substr(0, space));

    if (space == str.npos) {
      break;
    } else {
     str.remove_prefix(space + 1);
    }
  }

  return result;
}
```

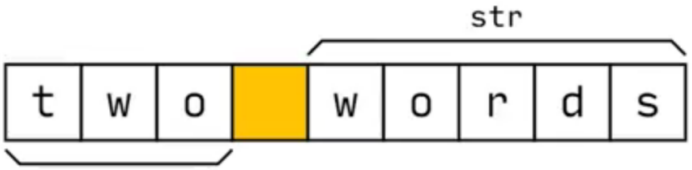
Разберём, как этот код работает. Пусть на вход подается строка `"two words"`.



Мы заходим в цикл, ищем пробел, он находится, его позиция равна трём. Далее вызывается метод `substr` от параметров `0`, `3`. Он выдаёт строку, которая начинается с нулевой позиции и имеет длину три. Это как раз первое слово.



После этого идёт проверка, не пора ли закончить цикл, откусывается префикс длины 4.



Далее переходим во вторую итерацию цикла, ищем пробел, он не находится. Пробела нет, `space` равно `npos`. Вызывается `substr` от нуля и `npos`, тем самым к ответу добавляется весь текущий `string_view`. Условие `space == str.npos` выполняется, цикл можно закончить.

Вернёмся к первому варианту `SplitIntoWordsView`. Что, если бы мы не создавали `string_view` по строке, а работали бы со строкой, назвав её `str`, вызывая методы `find` и `substr`? Такой код отработает, но не всё однозначно. Если разбить на слова более короткую строчку, например "a b", то в результате получим:

```
// a
// ?
```

Первый вариант функции, который мы использовали ещё во втором курсе, выводит букву `a`. А второй вариант, в котором мы работали со строкой, а не `string_view`, возвращает теперь какой-то символ. Проблема в том, что метод `substr` для строки создаёт новую строчку. Созданные временные объекты могут уничтожиться, если их никуда не сохранить. Получается, мы создали новую строку, сохранили в вектор `string_view` указатель на данные этой строки и тут же эта строка уничтожилась, потому что была временным объектом, то есть указатель `string_view` указывает в никуда. Если `string_view` ссылается на какой-то диапазон символов, следите, чтобы он не уничтожился.

Как сделать правильно и более компактно? Можно сразу в функции принять `string_view` по значению:

```
vector<string_view> SplitIntoWordsView(string_view str) {
  ...
```

```
}
```

Если вы хотите в функции работать со `string_view`, удобно принять в неё `string_view`, потому что вы не получите проблем с методом `substr`, более того, в функцию можно передавать как строку, так и `string_view`. Этот вариант универсальнее.