



Основы разработки на C++: красный пояс

Неделя 5

Move-семантика и базовая многопоточность



Оглавление

Move-семантика и базовая многопоточность	3
5.1 Move-семантика	3
5.1.1 Перемещение временных объектов.	3
5.1.2 Перемещение в других ситуациях	6
5.1.3 Функция <code>move</code>	8
5.1.4 Когда перемещение не помогает	11
5.1.5 Конструктор копирования и оператор присваивания	13
5.1.6 Конструктор перемещения и перемещающий оператор присваивания	15
5.1.7 Передача параметра по значению	18
5.1.8 Move-итераторы	18
5.1.9 Некопируемые типы	20
5.1.10 NRVO и copy elision	21
5.1.11 Опасности <code>return</code>	22
5.2 Базовая многопоточность	23
5.2.1 <code>async</code> и <code>future</code>	23

5.2.2	Задача генерации и суммирования матрицы	24
5.2.3	Особенности шаблона <code>future</code>	26
5.2.4	Состояние гонки	27
5.2.5	<code>mutex</code> и <code>lock_guard</code>	29
5.2.6	<code><execution></code> , которого нет	31

Move-семантика и базовая МНОГОПОТОЧНОСТЬ

5.1. Move-семантика

5.1.1. Перемещение временных объектов.

Рассмотрим задачу. Пусть есть функция, которая возвращает тяжёлую строку:

```
string MakeString() {  
    return string(100000000, 'a');  
}
```

Пусть необходимо добавить эту строчку в некоторый вектор строк.

```
int main() {  
    vector<string> strings;  
    string heavy_string = MakeString();  
    strings.push_back(heavy_string);  
  
    return 0;  
}
```

Мы хотим положить результат вызова функции `MakeString()` в вектор. Это можно сделать, не используя переменную `heavy_string`.

```
int main() {  
    vector<string> strings;  
    strings.push_back(MakeString());  
  
    return 0;  
}
```

Замерим время работы каждой реализации, используя макрос LOG_DURATION.

```
// with variable: 299 ms
// without variable: 168 ms
```

Второй вариант работает быстрее, потому что временный объект – результат вызова функции – не сохраняется в переменную.

Исследуем скорость работы алгоритмов, в которых в вектор добавляется временный объект, созданный с помощью конструктора строки.

```
vector<string> strings;
string heavy_string = string(100000000, 'a');
strings.push_back();

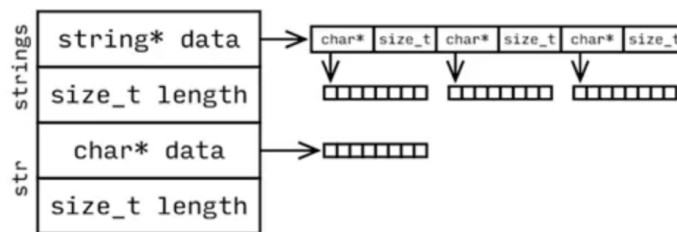
vector<string> strings;
strings.push_back(string(100000000, 'a'));

// ctor: with variable: 201 ms
// ctor: without variable: 122 ms
```

Без использования промежуточной переменной код работает быстрее. Во втором варианте в `push_back` передаётся временный объект, он забирает данные этого объекта, не копируя. Подробнее разберёмся, как это работает.

Рассмотрим процесс добавления в вектор переменной. Вспомним, как хранятся данные вектора и строки в памяти.

```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```

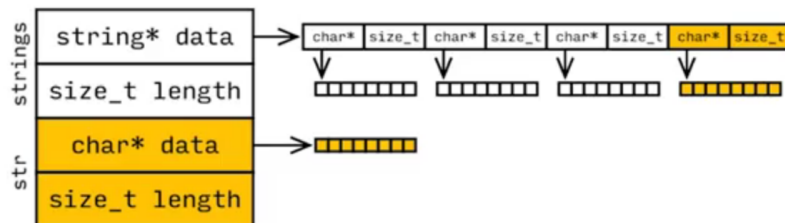


Локальная переменная `strings`, которая является вектором, и локальная переменная `str` со строкой хранятся на стеке. Вектор представляется указателем на свои данные в куче и длиной.

Кроме того, есть локальная переменная со строкой, которая представляет собой указатель на данные в куче и длину. У вектора в куче хранятся строки подряд, где каждая строка – указатель на свои данные и длина.

`push_back`, вызванный от локальной переменной, должен скопировать данные этой строки.

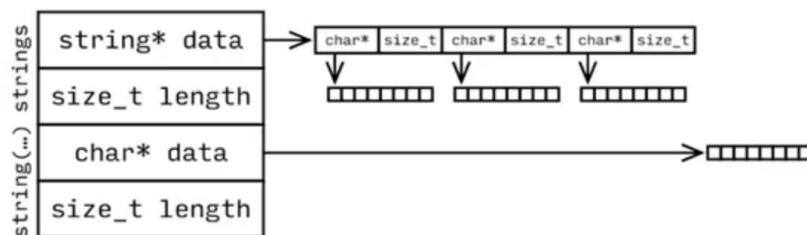
```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```



Выделяем память под восемь символов, в вектор кладём указатель на них.

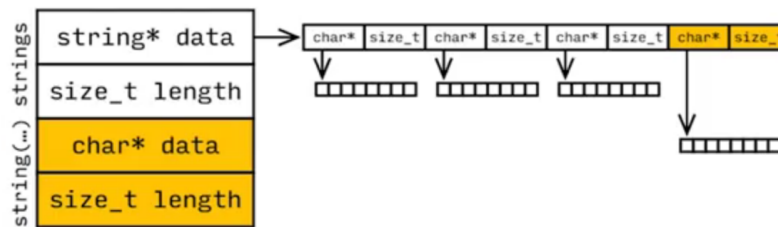
Если мы хотим добавить в вектор временный объект, то его данные в куче, на которые он ссылается, никому не будут нужны. Если их никто не заберёт, то они просто уничтожатся, потому что объект временный.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



`push_back` имеет право эти данные не копировать, а просто свой указатель, который добавится в данные вектора, направить на эти данные, а у той самой локальной временной строки эти данные отобразить.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



5.1.2. Перемещение в других ситуациях

Разберём ещё один класс ситуаций, когда у нас не происходит копирование данных временного объекта. В языке C++ везде, где может происходить копирование, имеет место и перемещение. Вспомним, где может происходить копирование объектов. Самый простой случай – это присваивание.

```
string target_string = "old_value";
string source_string = MakeString();
target_string = source_string;
```

В таком варианте здесь будет происходить копирование. Здесь тоже можно убрать промежуточную переменную `source_string` и сэкономить ресурсы на копировании.

```
string target_string = "old_value";
target_string = MakeString();
```

Замерим время работы программы с присваиванием в промежуточную переменную и без промежуточной переменной.

```
// assignment, with variable: 243 ms
// assignment, without variable: 119 ms
```

При использовании метода `set::insert` тоже можно обойтись без промежуточной переменной.

```
// set::insert, with variable
set<string> strings;
string heavy_string = MakeString();
strings.insert(heavy_string);
```

```
// set::insert, without variable
set<string> strings;
strings.insert(MakeString());

// set::insert, with variable: 217 ms
// set::insert, without variable: 100 ms
```

Теперь рассмотрим пример со словарём. Промежуточную переменную создаём и для ключа, и для значения.

```
map<string, string> strings;
string key = MakeString();
strings value = MakeString();
strings[key] = value;
// map::operator[], with variables: 474 ms
```

Избавимся от промежуточной переменной для значения.

```
map<string, string> strings;
string key = MakeString();
strings[key] = MakeString();
// map::operator[], with variable for key: 305 ms
```

Теперь избавимся и от промежуточной переменной для ключа.

```
map<string, string> strings;
strings[MakeString()] = MakeString();
// map::operator[], without variables: 210 ms
```

Напишем функцию MakeVector(), создающую вектор, и попробуем этот вектор куда-нибудь положить.

```
vector<int> MakeVector() {
    return vector<int>(100000000, 0);
}
```

Попробуем положить такой вектор во множество.

```
// set::insert for vector, with variable
set<vector<int>> vectors;
vector<int> heavy_vector = MakeVector();
vectors.insert(heavy_vector);
```



```
// set::insert for vector, without variable
set<vector<int>> vectors;
vectors.insert(MakeVector());

// set::insert for vector, with variable: 1062 ms
// set::insert for vector, without variable: 386 ms
```

Если у вас есть некоторый временный объект, старайтесь не потерять это важное свойство – временность. Так компилятор сэкономит на копировании.

5.1.3. Функция move

Есть ситуации, когда есть невременный объект, но про него известно, что там, где он создан, он больше не понадобится.

Есть задача – считать набор строк из потока ввода и эти строки положить в вектор. Это делает функция `ReadStrings()`.

```
vector<string> ReadStrings(istream& stream) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        strings.push_back(s);
    }
    return strings;
}
```

После того как мы кладем строчку `s` в вектор, она нам больше не понадобится. Хочется, чтобы метод `push_back` вел бы себя в этом случае как с временным объектом. Мы хотим изменить семантику объекта.

Есть функция `move`, которая может изменить семантику даже постоянного объекта. Чтобы использовать функцию `move`, нужно подключить модуль `utility`. Цикл теперь выглядит так:

```
while (stream >> s) {
    strings.push_back(move(s));
}
```

Проверим работу функции:

```
int main() {
    for (const string& s : ReadStrings(cin)) {
        cout << s << "\n";
    }
    return 0;
}
// ввод: Red belt C++
// вывод:
// Red
// belt
// C++
```

Разницу между реализациями с `move` и без можно определить не только по времени работы кода, но и по содержанию переменной `s`. Посмотрим, что лежит в переменной `s` до `push_back`'а и после него. Также посмотрим, что лежит в конце вектора.

```
while (stream >> s) {
    cout << "s = " << s << "\n";
    strings.push_back(s);
    cout << "s = " << s <<
        ", strings.back() = " << strings.back() << "\n";
}
// ввод: a b c
// вывод:
// s = a
// s = a, strings.back() = a
// s = b
```

Буква `a` скопировалась. Если добавить вызов `move` в метод `push_back`:

```
// ввод: a b c
// вывод:
// s = a
// s = , strings.back() = a
// s = b
```

Метод `push_back` забрал данные из строки `s`.

Покажем, что код ускоряется. Для этого напомним функцию, которая умеет как перемещать, так и не перемещать.

```
vector<string> ReadStrings(istream& stream, bool use_move) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        if (use_move) {
            strings.push_back(move(s));
        } else {
            strings.push_back(s);
        }
    }
    return strings;
}
```

Воспользуемся функцией `GenerateText()`. Она генерирует текст из миллиарда символов и разбивает его пробелами через каждые десять миллионов символов.

```
int main() {
    const string text = GenerateText();
    { LOG_DURATION("without move");
      istringstream stream(text);
      ReadStrings(stream, false);
    }
    { LOG_DURATION("with move");
      istringstream stream(text);
      ReadStrings(stream, true);
    }
}
// without move: 26359 ms
// with move: 17586 ms
```

Получили существенное ускорение за счет избавления от лишнего копирования.

Этим примером область применения функции `move` не ограничивается. Бывают ситуации, когда результат вызова функции разбиения строки на слова должен жить дольше чем исходная строка. Рассмотрим следующую реализацию функции `SplitIntoWords`:

```
vector<string> SplitIntoWords(const string& text) {
    vector<string> words;
    string current_word;
    for (const char c : text) {
        if (c == ' ') {
```

```
    words.push_back(current_word); // вызываем push_back от переменной, которая нам
    // дальше не нужна
    // логично будет обернуть её в move
    current_word.clear();
} else {
    current_word.push_back(c);
}
}
words.push_back(current_word);
return words;
}
```

5.1.4. Когда перемещение не помогает

Если у объекта нет данных в куче, а основные данные на стеке, то данные придётся копировать. Массив хранит свои данные на стеке. Продемонстрируем на его примере, что перемещение не помогает ускорить работу с ним.

```
const int SIZE = 10000;

array<int, SIZE> MakeArray() {
    array<int, SIZE> a;
    a.fill(8);
    return a;
}
```

Создадим вектор массивов и положим массив в вектор десять тысяч раз с использованием промежуточной переменной и без использования промежуточной переменной.

```
int main() {
    { LOG_DURATION("with variable");
      vector<array<int, SIZE>> arrays;
      for (int i = 0; i < 10000; ++i) {
          auto heavy_array = MakeArray();
          arrays.push_back(heavy_array);
      }
    }
    { LOG_DURATION("without variable");
      vector<array<int, SIZE>> arrays;
```

```
    for (int i = 0; i < 10000; ++i) {
        arrays.push_back(MakeArray());
    }
}

return 0;
}
// with variable: 1191 ms
// without variable: 1148 ms
```

Ускорения не произошло. Рассмотрим другой пример.

```
string MakeString() {
    return "C++";
}

int main() {
    vector<string> strings;
    string s = MakeString();
    cout << s << "\n";
    strings.push_back(s);
    cout << s << "\n";

    return 0;
}
// C++
// C++
```

Теперь сделаем переменную `s` константной:

```
const string s = MakeString();

// C++
// C++
```

Обернём строчку `s` в `move`:

```
strings.push_back(move(s));

// C++
// C++
```

Перемещения строки не произошло. Строка константная, перемещение из неё не работает.

Вызов `move` для константного объекта бесполезен. Следите за константностью перемещаемого объекта.

5.1.5. Конструктор копирования и оператор присваивания

Обсудим, что происходит в следующих ситуациях:

```
vector<int> source = /* ... */;  
vector<int> target = source;  
  
vector<int> source2 = /* ... */;  
target = source2;
```

Зачем это необходимо:

- Научиться перемещать собственные классы;
- Разговаривать с разработчиками на одном языке;
- Читать документацию и понимать, где может происходить перемещение;
- Развить интуицию относительно `move`-семантики.

В следующих случаях вызывается конструктор копирования (copy constructor):

```
vector<int> source = /* ... */;  
vector<int> target = source;  
vector<int> target2(source);
```

В следующих случаях вызывается оператор копирующего присваивания (copy assignment operator):

```
vector<int> source = /* ... */;  
vector<int> target = /* ... */;  
target = source;  
target.operator=(source);
```

Познакомимся поближе с этими методами на примере класса `Logger`, который будет логировать вызовы этих методов.

```
class Logger {
public:
    Logger() { cout << "Default ctor\n"; } // конструктор по умолчанию
    Logger(const Logger&) { cout << "Copy ctor"; } // конструктор копирования
    void operator=(const Logger&) { cout << "Copy assignment\n"; } // оператор
    // присваивания
};

int main() {
    Logger source; // вызывается конструктор по умолчанию
    Logger target = source; // вызывается конструктор копирования

    return 0;
}
// Default ctor
// Copy ctor

vector<Logger> loggers;
loggers.push_back(target);
// Copy ctor

source = target; // вызывается оператор присваивания
// Copy assignment
```

Для собственных типов при необходимости компилятор сам генерирует конструктор копирования и оператор присваивания, которые просто копируют все поля.

Для типов, которые самостоятельно управляют памятью, нужно самостоятельно реализовывать копирование и присваивание.

Напишем конструктор копирования для класса `SimpleVector`.

Добавим конструктор от константной ссылки на `SimpleVector`.

```
SimpleVector(const SimpleVector& other);
```

Реализуем этот конструктор:

```
template <typename T>
```

```
SimpleVector<T>::SimpleVector(const SimpleVector<T>& other)
: data(new T[other.capacity]),
  size(other.size),
  capacity(other.capacity)
{
    copy(other.begin(), other.end(), begin());
}
```

5.1.6. Конструктор перемещения и перемещающий оператор присваивания

В следующих случаях вызывается конструктор перемещения (move constructor):

```
vector<int> source = /* ... */;
vector<int> target = move(source);
vector<int> target2(move(target));

vector<vector<int>> vectors;
vectors.push_back(vector<int>(5));
```

В следующем случае инициализация временным объектом оптимизируется без вызова конструктора перемещения. Этот случай особый, его обсудим позже.

```
vector<int> MakeVector();

vector<int> target = MakeVector();
```

В следующих случаях вызывается оператор перемещающего присваивания (move assignment operator):

```
vector<int> source = /* ... */;
vector<int> target = /* ... */;
target = move(source);
target = vector<int>(5);
```


Если у вас есть собственный класс с конструктором копирования, но без конструктора перемещения, то компилятор делает перемещение эквивалентным копированию. Рассмотрим пример с классом `SimpleVector`. Скажем компилятору самостоятельно сгенерировать конструктор перемещения.

```
// rvalue reference
SimpleVector(const SimpleVector&& other) = default;
```

`rvalue reference` ведёт себя как обыкновенная ссылка, но позволяет принимать временные объекты.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 1 1
```

Такой код падает в конце, на деструкторах. Придётся самостоятельно реализовывать конструктор перемещения.

```
template <typename T>
SimpleVector<T>::SimpleVector(SimpleVector<T>&& other)
    : data(other.data),
      size(other.size),
      capacity(other.capacity)
{
    other.data = nullptr;
    other.size = other.capacity = 0;
}
// 0 1
```

Теперь у объекта, из которого мы перемещали, размер 0, а у объекта, в который мы перемещали – размер 1.

Реализуем оператор перемещающего присваивания.

```
void operator=(SimpleVector&& other);
```

Он будет реализован примерно как конструктор перемещения.

```
template <typename T>
void SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    delete[] data;
    data = other.data;
    size = other.size;
    capacity = other.capacity;

    other.data = nullptr;
    other.size = other.capacity = 0;
}
```

Проверим конструктор перемещения.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target(1);
    target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 0 1
```

Перегрузка по rvalue-ссылке – способ отличить временный объект от постоянного.

```
target = source;
// ↑ вызывается operator=(const vector<int>&)
target = vector<int>(5);
// ↑ вызывается operator=(vector<int>&&)
target = move(source);
// ↑ вызывается operator=(vector<int>&&)
```

При необходимости компилятор сам генерирует конструктор перемещения и оператор перемещающего присваивания, которые просто перемещают все поля. Если класс не управляет памятью самостоятельно, то перемещение для него будет просто работать.

5.1.7. Передача параметра по значению

Оптимизируем методы `ChangeFirstName` и `ChangeLastName` из класса `Person`, который реализовывался в задаче «Имена и фамилии-4»

```
void ChangeFirstName(int year, const string& first_name) {
    first_names[year] = first_name;
}
void ChangeLastName(int year, const string& last_name) {
    last_names[year] = last_name;
}
```

Необходимо принимать строки по значению, а не по ссылке. Внутри функции будем перемещать строку внутрь словаря.

```
void ChangeFirstName(int year, string first_name) {
    first_names[year] = move(first_name);
}
void ChangeLastName(int year, string last_name) {
    last_names[year] = move(last_name);
}
```

Возможны два случая. Если мы вызываем `ChangeFirstName` от временного объекта, то он проинициализирует переменную `first_name`, поскольку объект временный, то для него вызовется конструктор перемещения. Затем мы снова вызовем перемещение, переместим `first_name` внутрь контейнера. Будет два перемещения.

Если мы вызываем этот метод не от временного объекта, тогда этот объект скопируется в аргумент функции, затем случится перемещение этого объекта внутрь контейнера.

Итого, приняв параметр функции по значению, мы можем сделать его универсальным, вызывать как от временных объектов, так и от постоянных.

5.1.8. Move-итераторы

Рассмотрим конструкцию, позволяющую сделать использование `move`-семантики более простым.

Рассмотрим задачу `SplitIntoSentences`, в которой нужно было написать функцию, принимающую набор токенов, разбивающую их на предложения. Рассмотрим саму функцию `SplitIntoSentences`:

```
template <typename Token>
vector<Sentence<Token>> SplitIntoSentences(
    vector<Token>> tokens) {
    vector<Sentence<Token>> sentences;
    auto tokens_begin = begin(tokens);
    while (tokens_begin != tokens_end) {
        const auto sentence_end =
            FindSentenceEnd(tokens_begin, tokens_end);
        Sentence<Token> sentence;
        for (; tokens_begin != sentence_end; ++tokens_begin) {
            sentence.push_back(move(*tokens_begin));
        }
        sentences.push_back(move(sentence));
    }
    return sentences;
}
```

В ней мы заводим итератор `tokens_begin` и идём этим итератором по токенам, находим конец очередного предложения. Берем текущий токен, идем итератором от текущего токена до конца предложения, все эти токены вставляем в текущее предложение. Затем это предложение вставляем в вектор предложений.

Рассмотрим работу цикла `for`. Он перебирает токены в их диапазоне, каждый из них вставляет в вектор.

Подключим модуль `iterator`. После этого станет доступна специальная функция, которую мы вызовем от итераторов. Она называется `make_move_iterator`. Такая функция возвращает обёртку над итератором, которая, если мы хотим скопировать объект, перемещает его. Функция `make_move_iterator` меняет семантику итератора, чтобы при обращении к нему данные перемещались бы, а не копировались.

Используя `make_move_iterator`, можем заменить этот кусок...

```
Sentence<Token> sentence;
for (; tokens_begin != sentence_end; ++tokens_begin) {
    sentence.push_back(move(*tokens_begin));
}
sentences.push_back(move(sentence));
```

...на такой:

```
sentences.push_back(Sentence<Token>(
    make_move_iterator(tokens_begin),
    make_move_iterator(sentence_end)
));
```

Осталось в конце менять итератор `tokens_begin`:

```
tokens_begin = sentence_end;
```

Рассмотрим задачу «Считалка Иосифа». Рассмотрим следующий цикл:

```
for (uint32_t i = 0; i < range_size; ++i, ++range_begin) {
    *range_begin = move(permutation[i]);
}
```

Без цикла это можно записать так:

```
copy(
    make_move_iterator(begin(permutation)), make_move_iterator(end(permutation)),
    range_begin);
```

Алгоритм `move` будет перемещать данные, а не копировать. Это избавит от необходимости вызывать `make_move_iterator`.

```
move(begin(permutation), end(permutation), range_begin);
```

5.1.9. Некопируемые типы

Продemonстрируем, как можно сделать тип, который не будет уметь копировать на примере `Logger`'а. Нужно написать `delete` вместо тела конструктора копирования:

```
Logger(const Logger&) = delete;
```

В языке есть такие типы, которые копировать бессмысленно, например, потоки ввода и вывода. Посмотрим, как с ними обращаться на примере вектора потоков.

```
vector<ofstream> streams;
streams.reserve(5);
```

```
for (int i = 0; i < 5; ++i) {
    ofstream stream(to_string(i) + ".txt");
    stream << "File #" << i << "\n";
    streams.push_back(move(stream));
}
for (auto& stream : streams) {
    stream << "Vector is ready!" << endl;
}
```

Откроем, например, файл «0.txt». Там написано:

```
// File #0
// Vector is ready!
```

Получилось работать с файловыми потоками, несмотря на то, что это не копируемые объекты.

5.1.10. NRVO и copy elision

Можно оптимизировать копирование объектов, у которых данные только на стеке. Продемонстрируем это на примере Logger'a. Пусть у нас есть функция MakeLogger():

```
Logger MakeLogger() {
    return Logger(); // temporary -> returned temporary
}
```

Временным объектом Logger() мы инициализируем тот промежуточный объект, который должен вернуться из функции. В функции main мы из временного объекта, который вернулся из функции, инициализируем переменную:

```
int main() {
    Logger logger = MakeLogger(); // temporary -> variable

    return 0;
}
// Default ctor
```

Поскольку в обоих случаях новый объект инициализируется временным объектом, происходит перемещение.

В результате работы кода вызывался только конструктор по умолчанию, конструкторы копирования и перемещения не вызвались. Дело в том, что в некоторых случаях компилятор умеет оптимизировать перемещение. Например:

- при возвращении из функции временного объекта;
- при инициализации нового объекта временным объектом.

Такая оптимизация называется *copy elision*.

При возвращении локальной переменной из функции перемещение и копирование опускаются. Такая оптимизация называется *named return value optimization*.

5.1.11. Опасности `return`

Рассмотрим два случая, когда `return` оптимизируется не так хорошо, как в случаях из предыдущего пункта.

```
pair<ifstream, ofstream> MakeStreams(const string& prefix) {  
    ifstream input(prefix + ".in");  
    ofstream output(prefix + ".out");  
    return {input, output};  
}
```

Код не компилируется, компилятор не может составить пару потоков. `input` и `output` передаются в конструктор пары, затем созданная с помощью этого конструктора пара должна проинициализировать возвращаемый временный объект. Поскольку эта пара тоже временная, то инициализация временного объекта, возвращаемого из функции этой парой, происходит безболезненно, но передача переменных `input` и `output` в конструктор пары происходит по обычным правилам языка C++. Чтобы решить проблему, следует обернуть `input` и `output` в `move`.

Если функция должна вернуть некоторый объект, например поток ввода:

```
ifstream MakeInputStream(const string& prefix) {  
  
}
```

Внутри функции получили пару объектов, вернуть ходим только один её элемент.

```
ifstream MakeInputStream(const string& prefix) {  
    auto streams = MakeStreams(prefix);  
    return streams.first;  
}
```

`streams.first` не является временным объектом. Также это выражение не является названием локальной переменной – это поле локальной переменной.

Проблема решается, если обернуть `streams` в `move`.

5.2. Базовая многопоточность

5.2.1. `async` и `future`

Напишем функцию, которая будет суммировать элементы двух векторов. В однопоточной синхронной версии наша функция будет выглядеть так:

```
int SumToVectors(const vector<int>& one,  
                const vector<int>& two) {  
    return accumulate(begin(one), end(one), 0)  
        + accumulate(begin(two), end(two), 0);  
}
```

В начале мы находим сумму элементов одного вектора, потом сумму элементов другого вектора. Мы могли бы один вектор суммировать асинхронно, другой вектор суммировать одновременно с первым, потом сложить результаты. Понадобится заголовочный файл `future`:

```
#include <future>
```

Вызываем функцию `async`, она запускает асинхронную операцию. В данном случае возвращается результат суммирования вектора `one`.

```
future<int> f = async([] {  
    return accumulate(begin(one), end(one), 0);  
});
```

Далее в переменную `result` присваиваем сумму элементов второго вектора.


```
int result = accumulate(begin(two), end(two), 0);
```

Далее возвращаем `result` плюс то, что возвращает `async`. `async` возвращает `future`.

```
return result + f.get();
```

Такой код не компилируется, потому что не захвачена переменная `one`.

```
future<int> f = async([&one] {  
    return accumulate(begin(one), end(one), 0);  
});
```

Когда мы пишем `one` в квадратных скобках лямбды, то происходит копирование внутрь лямбда-функции. Чтобы он не копировался, его следует передавать по ссылке.

5.2.2. Задача генерации и суммирования матрицы

У нас есть матрица:

```
vector<vector<int>> matrix;
```

Она генерируется с помощью функции `GenerateSingleThread`:

```
vector<vector<int>> GenerateSingleThread(size_t size) {  
    vector<vector<int>> result(size);  
    GenerateSingleThread(result, 0, size);  
    return result;  
}
```

Функция вызывает другую функцию `GenerateSingleThread`, которая является шаблоном.

```
template <typename ContainerOfVectors>  
void GenerateSingleThread(  
    ContainerOfVectors& result;  
    size_t first_row,  
    size_t column_size  
) {  
    for (auto& row : result) {  
        row.reserve(column_size);
```

```

    for (size_t column = 0; column < column_size; ++column) {
        row.push_back(first_row ^ column);
    }
    ++first_row;
}
}

```

Далее матрица обрабатывается с помощью функции `SumSingleThread`.

Запустим программу:

```

int main() {
    LOG_DURATION("Total");
    const size_t matrix_size = 7000;

    vector<vector<int>> matrix;
    {
        LOG_DURATION("Single thread generation");
        matrix = GenerateSingleThread(matrix_size);
    }
    {
        LOG_DURATION("Single thread sum");
        cout << SumSingleThread(matrix) << endl;
    }
}

// Single thread generation: 1254 ms
// Single thread sum: 375 ms
// 195928050144
// Total: 1651 ms

```

Мы хотим ускорить программу. Попробуем генерировать матрицу многопоточно. Напишем функцию `GenerateMultiThread`. Она будет принимать `page_size` 0 – желаемый размер страницы, который будет передаваться потоку.

```

vector<vector<int>> GenerateMultiThread(
    size_t size, size_t page_size
) {
    vector<vector<int>> result(size);
    return result;
}

```

Мы хотим разбивать вектор `result` на несколько частей. Здесь подходит шаблон `Paginator`.

```
vector<vector<int>> GenerateMultiThread(
    size_t size, size_t page_size
) {
    vector<vector<int>> result(size);
    vector<future<void>> futures;
    size_t first_row = 0;
    for (auto page : Paginator(result, page_size)) {
        futures.push_back(
            async([page, first_row, size] {
                GenerateSingleThread(page, first_row, size);
            })
        );
        first_row += page_size;
    }

    return result;
}
```

Теперь в `main()` будем вызывать многопоточный генератор.

```
{
    LOG_DURATION("Multi thread generation");
    matrix = GenerateMultiThread(matrix_size, 2000);
}

// Single thread generation: 1229 ms
// Multi thread generation: 611 ms
// Single thread sum: 365 ms
// 195928050144
// Total: 2345 ms
```

Многопоточная генерация матрицы оказалась в два раза быстрее, чем однопоточная.

5.2.3. Особенности шаблона `future`

Обратим внимание на вектор `futures` из функции `GenerateMultiThread`. Мы его объявили, сложили в него результаты вызова `async` и больше его не вызывали. Сохраняя результаты в вектор,

мы откладываем вызов их деструктора, в котором вызывается `get()`. Если результат вызова функции `async` не сохранить в переменную, то программа может выполняться последовательно.

5.2.4. Состояние гонки

Разработаем класс `Account`:

- Он представляет собой банковский счёт;
- Не должен позволять тратить больше денег, чем есть на счету;
- Не должен допускать, чтобы баланс счёта стал отрицательным.

```
struct Account {  
    int balance = 0;  
  
    bool Spend(int value) {  
        if (value <= balance) {  
            balance -= value;  
            return true;  
        }  
        return false;  
    };  
};
```

Напишем функцию, которая будет пытаться 100000 раз потратить один рубль. Она возвращает количество потраченных денег.

```
int SpendMoney (Account& account) {  
    int total_spent = 0;  
    for (int i = 0; i < 100000; ++i) {  
        if (account.Spend(1)) {  
            ++total_spent;  
        }  
    }  
    return total_spent;  
}
```

```
int main() {
    Account my_account(100000);

    cout << "Total spent: " << SpendMoney(my_account)
          << "Balance: " << my_account.balance << endl;
}
// Total spent: 100000
// Balance: 0
```

Пусть теперь несколько людей тратят деньги с семейного счёта асинхронно.

```
int main() {
    Account family_account(100000);

    auto husband = async(SpendMoney, ref(family_account));
    auto wife     = async(SpendMoney, ref(family_account));
    auto son      = async(SpendMoney, ref(family_account));
    auto daughter = async(SpendMoney, ref(family_account));

    int spent = husband.get() + wife.get() + son.get()
               + daughter.get();

    cout << "Total spent: " << spent << endl
          << "Balance: " << family_account.balance << endl;
}
// Total spent: 140573
// Balance: 0
```

Семья потратила больше денег, чем изначально лежало на счёте.

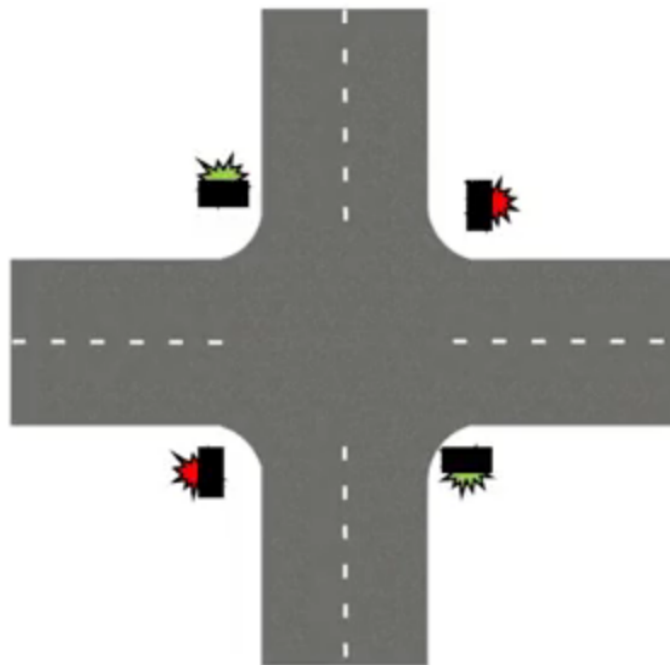
Если несколько потоков обращаются к одной и той же переменной, целостность данных может быть нарушена. Класс `Account` поддерживал инвариант: баланс никогда не становится отрицательным, мы не можем потратить больше, чем есть на счету. Этот инвариант был нарушен при одновременном обращении к данным. В этом заключается гонка данных. Чтобы её избежать, необходимо выполнять синхронизацию доступа к данным из нескольких потоков.

5.2.5. mutex и lock_guard

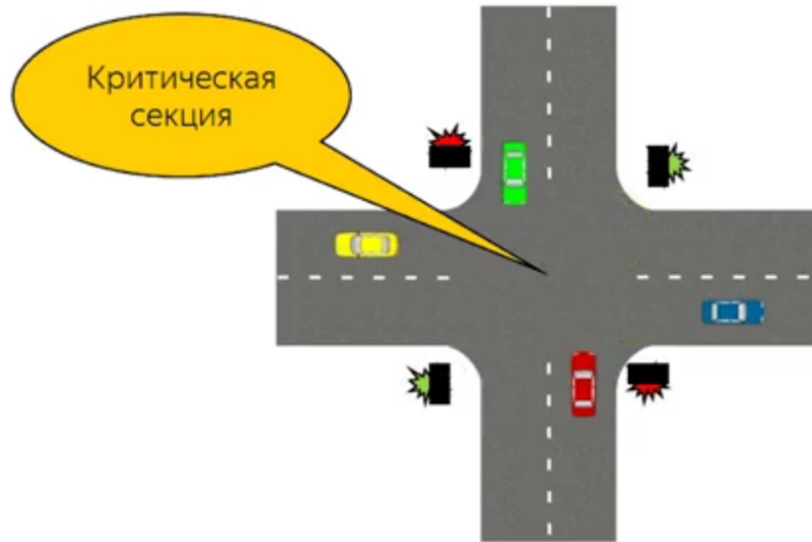
Добавим в наш класс Account `vector<int> history`, который будет сохранять все транзакции.

```
struct Account {  
    int balance = 0;  
    vector<int> history;  
  
    bool Spend(int value) {  
        if (value <= balance) {  
            balance -= value;  
            history.push_back(value);  
            return true;  
        }  
        return false;  
    };  
};
```

Текущий код нужно адаптировать для работы нескольких потоков, используя Mutex (MUTual EXclusion). Простым примером мьютекса является перекрёсток.



Мьютекс защищает критическую секцию. В программировании это тот участок кода, который в любой момент времени может выполнять не более одного потока.



Применим мьютекс в нашей программе. Подключим заголовочный файл `mutex`. В классе `Account` объявим поле

```
mutex m;
```

Критической секцией в нашем коде является метод `Spend`. Его нужно защитить мьютексом.

```
bool Spend(int value) {
    lock_guard<mutex> g(m);
    if (value <= balance) {
        balance -= value;
        history.push_back(value);
        return true;
    }
    return false;
};
```

Теперь программа работает правильно:

```
// Total spent: 100000
// Balance: 0
```

5.2.6. <execution>, которого нет

Вернёмся к примеру с генерацией и суммированием элементов матрицы. Изменим реализацию функции `GenerateSingleThread`. Заменим цикл `for` на алгоритм `for_each`. По сути он делает то же самое.

```
void GenerateSingleThread(
    ContainerOfVectors& result;
    size_t first_row,
    size_t column_size
) {
    for_each (
        begin(result),
        end(result),
        [&first_row, column_size] (vector<int>& row) {
            row.reverse(column_size);
            for (size_t column = 0; column < column_size; ++column) {
                row.push_back(first_row ^ column);
            }
            ++first_row;
        }
    );
}
```

В стандарте C++17 были введены параллельные версии стандартных алгоритмов. Чтобы получить параллельную версию алгоритма `for_each` достаточно подключить заголовочный файл `execution` и в качестве первого параметра в функции указать `execution::par`.

```
for_each (execution::par, ...)
```

Ни один из ведущих компиляторов на момент записи видео (апрель 2018) не реализовал поддержку параллельных версий алгоритмов. Сейчас этим воспользоваться нельзя.