Основы разработки на C++. Функции: принципы понятного кода

Оглавление

Зачем нужны функции?

1.1 Зачем нужны функции?

Что такое хороший код? Это очень субъективное понятие, и все зависит от контекста, если у вас маленькая команда и маленький проект, то можете писать более-менее как угодно, но если у вас в команде 20 человек, они пишут один и тот же код, то очень важно, чтобы все понимали его одинаково и придерживались одинаковых договоренностей. Поэтому вам будут даны рекомендации по написанию хорошего кода. Вы можете им следовать, а можете не следовать, но при этом вы будете понимать последствия нарушения этих рекомендаций. Итак, прежде чем приступить к обсуждению того, как же писать хорошие функции, мы начнем с обсуждения того, зачем же вообще в языке C++ нужны функции.

Для этого давайте возьмем задачу из второго курса — «Демографические показатели». Вот ее решение, которое было опубликовано.

```
enum class Gender {
    FEMALE,
    MALE
};

struct Person {
    int age;
    Gender gender;
    bool is_employed;
};

template <typename InputIt>
int ComputeMedianAge(InputIt range_begin, InputIt range_end) {
    if (range_begin == range_end) {
      return 0;
    }
    vector<typename InputIt::value_type> range_copy(range_begin, range_end);
    auto middle = begin(range_copy) + range_copy.size() / 2;
    nth_element(
        begin(range_copy), middle, end(range_copy),
        [](const Person& lhs, const Person& rhs) {
          return lhs.age < rhs.age;
        }
    );
```

```
    return middle->age;
}

void PrintStats(vector<Person> persons) {
    auto females_end = partition(
        begin(persons), end(persons), [](const Person& p) {
            return p.gender == Gender::FEMALE;
        }
    );
    auto employed_females_end = partition(
        begin(persons), females_end, [](const Person& p) {
            return p.is_employed;
        }
    );
    auto employed_males_end = partition(
        females_end, end(persons), [](const Person& p) {
            return p.is_employed;
        }
    );

    cout << "Median age = "
         << ComputeMedianAge(begin(persons), end(persons)) << endl;
    cout << "Median age for females = "
         << ComputeMedianAge(begin(persons), females_end) << endl;
    cout << "Median age for males = "
         << ComputeMedianAge(females_end, end(persons)) << endl;
    cout << "Median age for employed females = "
         << ComputeMedianAge(begin(persons), employed_females_end) << endl;
    cout << "Median age for unemployed females = "
         << ComputeMedianAge(employed_females_end, females_end) << endl;
    cout << "Median age for employed males = "
         << ComputeMedianAge(females_end, employed_males_end) << endl;
    cout << "Median age for unemployed males = "
         << ComputeMedianAge(employed_males_end, end(persons)) << endl;
}

int main() {
    int person_count;
    cin >> person_count;
    vector<Person> persons;
    persons.reserve(person_count);
    for (int i = 0; i < person_count; ++i) {
        int age, gender, is_employed;
        cin >> age >> gender >> is_employed;
        Person person{age, static_cast<Gender>(gender), is_employed == 1};
        persons.push_back(person);
    }
```

```
    PrintStats(persons);
    return 0;
}
```

А теперь давайте представим, что нам запретили пользоваться функциями, ну, кроме разве что функции `main`, и весь код оказался в функции `main`. Что тогда с этим кодом случится?

Давайте перейдем в конструктив и поймем, что же в этом коде такого плохого — что случилось от того, что мы избавились от функций? Итак,

- Во-первых, совершенно очевидно, что копируя код вычисления медианного возраста несколько раз, мы явно закрыли себе возможность переиспользования этого кода. Если мы где-то еще захотим вычислить медианный возраст набора людей, мы этот код должны скопировать и как-то его поменять, при этом высока вероятность ошибиться, конечно. То есть первое **преимущество функции** — это **возможность переиспользовать код**.

- Дальше, давайте ещё посмотрим, что же плохого в этом коде? Как мы будем его тестировать? Сейчас у нас огромнейшая функция `main`, и если мы хотим этот код протестировать, мы должны написать какую-то внешнюю программу, которая будет нашу запускать — подавать ей на вход входные данные в стандартный поток ввода и смотреть, что же она вывела в стандартный поток вывода. То есть на вход — набор людей, на выходе — несколько чисел. Мы так не сможем протестировать непосредственно функцию вычисления медианного возраста. Когда же эта функция была, мы могли написать на нее unit-тест, например, с помощью нашего unit-тест фреймворка, подав конкретные входные данные в функцию и посмотрев, что же она вычислила. Итак, второе **преимущество функций** — это **возможность писать на них unit-тесты**.

- Третье **преимущество функций** — это **отсутствие необходимости или даже минимальная необходимость в комментариях**. Потому что, если посмотреть на то, как выглядел хороший код с функцией `ComputeMedianAge`, тут было явно видно, что мы вычисляем медианный возраст от такого диапазона людей. В плохом же коде — это просто некоторый блок кода. То есть понятное название функции помогает нам понять, что же в этом коде происходит — она документирует этот код.

- Наконец, смежное преимущество — это фиксированный ввод и вывод функции. Когда функция есть, вот функция `ComputeMedianAge`, мы явно видим, что она принимает на вход — в данном случае два итератора на людей, и явно видим, что она возвращает — она возвращает целое число. Если же мы посмотрим соответствующий блок кода в плохом варианте, то будет совершенно неочевидно, что же здесь приходит на вход этого блока, а что же на выходе — совершенно непонятно, какими данными манипулирует этот участок кода. Итого, четвертое **преимущество функции** — это **фиксированная сигнатура**. Функция явно декларирует свой ввод и вывод.

1.2 Функции или методы классов?

Давайте обсудим разницу между функциями и методами класса. Итак, есть, во-первых, понятные технические отличия. У класса всё-таки есть поля, и любые методы имеют доступ к этим

полям, возможно, даже на запись, если эти методы не константные. Поэтому в эти поля можно уносить некоторый глобальный контекст методов класса. Соответственно, если у вас есть набор функций, и им постоянно нужно менять какие-то определённые данные или их читать, то вы можете сделать их методами класса, а весь глобальный контекст увести в поля этого класса. Это, с одной стороны, удобно, а, с другой стороны, конечно, может усложнять понимание, потому что метод, получается, ещё имеет какие-то дополнительные данные в полях, и неизвестно, что вообще с ними будет происходить.

С другой стороны, есть важное семантическое отличие, смысловое. А именно, в C++ принято, что какой-то конкретный класс или, вообще говоря, любой тип олицетворяет собой какой-то объект. Поэтому просто так объединить набор функций в какой-то класс просто потому, что вам это удобно, может быть довольно сомнительным действием. Давайте рассмотрим пример. Опять же, задача «Демографические показатели». Если написать некоторые функции, то функция `main` будет устроена предельно просто

```
int main() {
    PrintStats(ComputeStats(ReadPersons()));
    return 0;
}
```

У вас может возникнуть желание объединить функции с похожим смыслом, с похожим назначением в один клас или структуру. Структуру, скажем, `StatsManager`, и в неё внести все эти функции в виде методов.

```
struct StatsManager {
    static vector<Person> ReadPersons(istream& in_stream = cin);
    static AgeStats ComputeStats(vector<Person> persons);
    static void PrintStats(const AgeStats& stats, ostream& out_stream = cout);
};
```

Теперь давайте обновим функцию `main`. И здесь мы тоже должны к вызову каждого метода добавить `StatsManager::`.

```
int main() {
    StatsManager::PrintStats(
        StatsManager::ComputeStats(StatsManager::ReadPersons()));
    return 0;
}
```

И, в принципе, наверное, ради этого вы могли это сделать. Потому что сейчас явно видно, что все эти функции относятся к задаче работы со статистикой.

Но, с другой стороны, если мы всё делали ради того, чтобы добавить какую-то строчку и два двоеточия перед всеми функциями, какой-то общий префикс, который их как-то объединяет, почему бы нам не использовать пространство имён. С тем же успехом я мог весь этот код заключить в одноимённый `namespace`. Ну и получилось бы примерно то же самое, зато никаких лишних сущностей.

Итак, основная рекомендация: если вы хотите просто так объединить набор смежных функций в класс, не имеющий никаких полей, остановитесь и используйте пространство имён. Если же вы

делаете класс, потому что считаете, что в дальнейшем появится глобальный контекст, нет, не плодите, пожалуйста, лишние сущности и всё равно используйте пространство имён или просто свободные функции, в них нет ничего плохого.

1.3 Какими должны быть функции?

Какими же важными свойствами функции должны обладать, чтобы считаться хорошими, и чтобы код был понятным?

1. Первое свойство — это **понятность сигнатуры функции**. Вы должны уметь посмотреть на сигнатуру функции и из этого понять, что же функция делает. По ее названию, по входным параметрам, по типу выходного параметра.

2. Во-вторых, эта **функция должна быть понятна в месте вызова**. То есть когда функция вызывается, должно быть понятно примерно, что каждый параметр означает и чего ожидать от этой функции в привязке к этим параметрам.

3. **Функции надо писать такими, чтобы их было удобно тестировать**. Понятно, что есть функции, которые вы вообще не протестируете, но и написать функцию можно настолько плохо, что тестировать ее будет неудобно.

4. **Функция должна быть поддерживаемой**. То есть если придет какой-то другой разработчик, или вы, и вы захотите как-то обновить функциональность этой функции, расширить ее, это должно быть легко.

5. И наконец, **связанное свойство — это сфокусированность функции**. То есть функция должна решать одну конкретную задачу. Не несколько маленьких как-то переплетенно, а одну конкретную.

И давайте мы сейчас все эти важные свойства проиллюстрируем несколько гипертрофированными, но тем не менее, понятными примерами на примере известных вам уже задач.

Начнем с **понятности сигнатуры**. Задача «Экспрессы». Представьте, что вы увидите чье-то решение этой задачи и там есть функция `Process`.

```
void Process() {...}
```

Совершенно непонятно, что делает эта функция, если не посмотреть внутрь этой функции.

Еще один пример функции с плохой, непонятной сигнатурой — вот такой. Задача «Трекер задач». Опять же, представьте, что вы видите чье-то решение этой задачи, и там вот такая функция — `PerformPersonTasks`, которая принимает, во-первых, словарь из строки в `TasksInfo`, по неконстатной ссылке, затем она принимает имя человека по константной ссылке — это, допустим, понятно. Она принимает количество задач, которые надо выполнить, и еще две не константных ссылки `first` и `second` на объекты типа `TasksInfo`.

```
void PerformPersonTasks(map<string, TaskInfo>& person_tasks, const string& person, int
   task_count, TaskInfo& first, TaskInfo& second) {...}
```

Понятно ли, что это за `first` и `second`?

Еще какая-то не константная ссылка на `person_tasks`. То есть функция как минимум меняет данные по трем ссылкам — это первый аргумент функции и два последних. При этом она еще ничего не возвращает. То есть она принимает что-то на вход, а на самом деле, `TasksInfo` — это тоже словари, и как-то она эти три словаря меняет. Кошмар. Непонятно, что происходит.

На самом деле, изначально решение было довольно приятным, с приятным интерфейсом. Там был класс `TeamTasks`, у него было приватное поле `person_tasks_` с тем самым словарем и метод — `PerformPersonTasks`, который принимал имя человека и количество задач и возвращал через кортеж два словаря задач со статистикой — задачи, которые сделаны, и задачи, которые не затронуты.

```
class TeamTasks {
public:
    ...

    tuple<TaskInfo, TaskInfo> PerformPersonTasks(const string& person, int task_count);

private:
    map<string, TaskInfo> person_tasks_;
};
```

Вот такой интерфейс был понятным.

Второе важное свойство функции — это **понятность** этих функций **в месте вызова**. Давайте для примера возьмем задачу «Hotel manager». Представьте, что мы хотим вызвать метод `Book` с какими-то конкретными значениями с целочисленных полей, и у нас получается такая вот строка

```
manager.Book(0, "Mariott", 1, 2);
```

Понятно ли чем здесь отличается 1, 2 и 0 — какие-то 3 числа? Их можно совершенно спокойно перепутать, и код все равно продолжит компилироваться. Здесь могла бы помочь среда разработки, если навести на метод `Book`, можно увидеть, что у меня сначала `client_id`, потом `room_count`. Но ничто не мешает ошибиться при написании этой функции, не посмотрев подсказку, или просто не понять при чтении этого кода, что же здесь происходит. Итак, будьте осторожны с функциями, которые принимают несколько целочисленных аргументов.

Следующая проблема, это **удобство тестирования функций**. Вернемся к нашему замечательному примеру с функцией `Process` в экспрессах. Это функция не принимает ничего и не возвращает ничего.

```
void Process() {
    ...
    cin >> ...
    ...
    cout << ...
}
```

Как написать на нее unit-тест — непонятно. Непонятно, что вообще с ней делать, как для нее переопределить входные данные. Если бы она хотя бы принимала поток по ссылке, можно было

бы его переопределить, то еще куда ни шло, но сейчас глобальная переменные — потоки `cin` и `cout` не позволяют написать нормальный unit-тест на эту функцию.

Следующее свойство — это **поддерживаемость функции**. Давайте рассмотрим пример неподдерживаемой функции, даже неподдерживаемой архитектуры класса, на примере системы бронирования отелей.

Давайте представим, что внутри всех классов мы решили отказаться от структуры `Booking`.

```
struct Booking {
    int64_t time;
    int client_id;
    int room_count;
};
```

У нас теперь 3 отдельные очереди вместо одной: есть очередь времен, очередь `client_id` и очередь количеств комнат, и везде нужно писать три раза `push`, потом надо написать три раза `pop` — и, наверное, такой код может показаться нормальным, но как только мы будем добавлять какое-то новое свойство бронирования, нужно будет добавить еще одну очередь, `push`, и, самое важное — то место, где очень легко ошибиться, это забыть добавить `pop`. То есть, **структура нам здесь помогала сделать код более поддерживаемым, более устойчивым к ошибкам, которые могут возникнуть при расширении функциональности**.

И наконец про **сфокусированность**. Опять же хороший пример несфокусированной функции — это функция `Process`, потому что она делает сразу несколько вещей. Во-первых, она считывает данные, во-вторых, она выводит данные, ну и наконец она взаимодействует с переменной `routes`. Получается, что смотря на какой-то произвольный кусок функций, давайте представим, что она очень большая, нельзя заранее угадать, а не считает ли она ещё какие-то данные. Она может быть даже уже что-то считала и вывела, и потом еще раз что-то считала и вывела, то есть несколько задач одной функции в ней как-то могут перемещаться — и это очень неприятно и очень мешает понимать функции, особенно если они довольно большие.

Вы можете увидеть несколько примеров хороших функций, чтобы вы понимали на что вам ориентироваться.

Функция, которая вычисляет остаток от деления двух вещественных чисел.

```
double remainder(double x, double y);
```

Метод вектора `size`

```
template<typename T>
size_t vector<T>::size() const;
```

функция `to_string`, которая принимает число и возвращает строчку.

```
string to_string(int value);
```

Алгоритм `count`, который подсчитывает сколько в данном диапазоне элементов встречается конкретных объектов.

```
template<typename InputIt, typename T>
size_t count(InputIt first, InputIt last, const T& value);
```

1.4 Философия понятного кода

Мы довольно много поговорили о том, какими же должны быть хорошие функции. И наверняка у некоторых из вас начал зарождаться вопрос, почему же мы такие зануды? На самом деле мы, команда курса, считаем, что особенно когда вы работаете над большим проектом, в большой команде нужно максимально щепетильно относиться к качеству вашего кода. Когда вы проектируете интерфейс вашего кода, нужно быть параноиком и перфекционистом. **Очень важно выработать навык предвидения того, как можно неправильно интерпретировать ваш замысел, который вы вложили в ваш код**.

И вы здесь можете спросить: зачем мне быть идеальным разработчиком? Почему мне нельзя быть просто хорошим? И вот без этого всего, без всех этих страшных рекомендаций, просто буду писать код как-нибудь и все будет нормально.

Хорошо, представьте, что вы работаете в большой команде и вы хороши, но так процентов на 60. И вот у вас есть коллеги, скажем 10 человек, вы написали какой-то код и дальше ваши коллеги иногда приходят и его как-то читают, исправляют. И вот 6 человек, они правильно поняли ваш замысел, как-то код обновили, все довольны, вы довольны и коллеги довольны, код понятный, но остальные 4 ничего не поняли. Каждый из них либо прочитал код и страдает, либо как-то его кое-как исправил и все разнес, и в итоге страдают и эти 4 человека, и вы страдаете, потому что ваш код теперь совершенно непонятен и тем 6 людям тоже, им сначала было хорошо, а теперь они тоже в печали, потому что непонятно что случилось с вашим кодом. Именно поэтому планка качества к коду очень высока, особенно когда вы работаете в большой команде.

Однако, некоторые из вас могут сказать: зачем мне писать понятный код, если всегда есть комментарии? То есть написал код как-нибудь, прокомментировал, и все замечательно. Даже непонятный код будет понятен, потому что есть комментарии. Но знаете если вы издаете книгу с вашим кодом, то есть вы написали, все там пояснили, распечатали и она зафиксирована, ну почему бы и нет, пишите как хотите, но всё таки, если речь идет о реальном коде, который работает в продакшене, его как-то меняют, он как-то развивается, то нужно думать еще и о том, что ваш код будет кто-то менять. И вряд ли ему будет дело до изменения комментариев. Вообще главная проблема комментариев в том, что если вы забыли их обновить, то код продолжит компилироваться и нормально работать. Если же вы допустили ошибку в коде, то он сломается. **Поэтому комментарии часто рассинхронизируются с основным кодом и нужно стараться их количество минимизировать, например, за счет понятности функций**.

В некоторых командах, на самом деле, есть хорошая практика — документирование не совсем всего кода, а только заголовка функции, только сигнатуры, то есть там буквально, в нескольких строчках написано, что эта функция делает. Очень замечательно если ваша команда к этому приучена и обновляет эту документацию, но если команда так не делает, то довольно тяжело будет заставить ребят обновлять комментарии к функциям. Гораздо проще научиться все таки сами функции поддерживать в адекватном состоянии, поэтому комментарии, они иногда нужны если речь идет о сложном коде, но далеко не везде. **Старайтесь в первую очередь делать код понятным**.

Детали проектирования функций

2.1 Как передать объект в функцию

Как же передать в функцию один объект, точнее, как его принять?

Вопрос первый: **по значению или по константной ссылки надо принять этот объект**? Во-первых, стоит понимать, что достаточно легкие объекты дешевле скопировать, чем принять по ссылке, а потом работать со ссылкой. Например, целые числа.

И здесь правило такое: **если размер объекта не больше 16 байт, то его дешевле скопировать**. Здесь правда есть ловушка, если у вас есть совершенно произвольная структура и ее размер пока что 16 байт, то пусть вас это не обманывает, возможно в ней потом появятся другие поля и размер станет больше чем 16 байт. Поэтому вы можете принимать по значению только те легкие объекты, размер которых именно такой по смыслу, и никогда не изменится. Например, целое число.

Еще один случай, когда **надо принимать по значению, если вы хотите этот объект, его данные скопировать, и как-то их менять**.

Давайте рассмотрим несколько примеров

```
vector<Query> ReadQueries(int query_count);
```

`string_view` тоже можно принимать по значению, потому что это всегда 16 байт.

```
bool CheckName(string_view name);
```

```
bool CheckBooking(const Booking& booking);
```

```
vector<int> BuildSorted(vector<int> numbers);
```

Что если ваша функция хочет принимать не все параметры всегда? Например,

```
vector<Query> ReadQueries(int query_count, ReadMode mode = ReadMode::Normal);
```

В этом случае нас спасает конечно значения по умолчанию — вы их прекрасно знаете.

Другой случай, если объект, который вы хотите сделать необязательным, принимался по константной ссылке, или просто по ссылке, и вы хотите иметь возможность эту ссылку не передавать, вот здесь такой интересный момент, что можно эту ссылку сделать указателем и принимать объект по указателю.

```
vector<Query> ReadQueries(int query_count, ReadSettings* settings = nullptr);
```

Понятно, что в современном C++, если не вспоминать о том, что указателем можно ходить по памяти, указатель отличается от ссылки только тем, что он может быть нулевым. Именно это свойство здесь можно использовать. То есть вы можете принять объект по указателю, а не по ссылке, и дать этому указателю значение по умолчанию `nullptr`, и дальше можно внутри функции это обработать. Если указатель нулевой, значит объект не передали.

Еще один важный момент: когда вы читаете сигнатуру каких-то функций, как правило в документации, вы можете видеть там двойные ссылки перед параметрами. Вы уже знаете, что например, как в конструкторе перемещения у вектора, это означает, что вот конкретно этот вариант этого конструктора принимает обязательно временный вектор, ну и дальше данные из него перемещает.

```
vector<T>::vector<T>(vector<T>&& other);
```

Но есть и другой интересный случай, когда вы можете увидеть два амперсанда в документации, например

```
template<typename M>
/*...*/ map<K, V>::insert_or_asiign(const K& k, M&& obj);
```

Пока просто запомните, что такая конструкция, когда у вас есть шаблонный тип и потом два амперсанда, означает, что эта функция может принять как временный объект, так и обычный, постоянный объект, и если там обычный объект, она не будет из него забирать данные, ну и не сможет, понятное дело, а если временный, то заберет.

Наконец, еще один важный нюанс, если речь идет о методах классов, то `this`, указатель на текущий объект по сути является неявным параметром метода, и нужно понимать, что если вы видите какой-то **метод, он принимает данные не только из своих параметров, но еще и из полей класса**, в частности из этого следует, что если у класса очень много полей, то они все, все эти поля являются по сути входными параметрами для всех методов, и если у параметров методов много, и у класса много полей, то получается перегруженность с большим количеством входных объектов.

Таким образом, передавать объект через глобальные переменные плохо, потому что непонятно и нетестируемо. Можно передать по назначению, если легко скопировать, то есть 16 байт или меньше, причем не только сейчас, но и всегда в будущем, можно передать по контактной ссылке.

Если вы хотите, чтобы можно было параметр не передавать, используйте значение по умолчанию.

Если вы хотите передавать в функцию ссылку или ничего, то есть сделать ссылку не обязательно, то сделать ее указателем, со значением по умолчанию `nullptr`.

И наконец, просто имейте в виду, что `this` это неявный параметр любого метода.

2.2 Как передать в функцию набор объектов

Давайте обсудим, как передать в функцию много объектов, если они имеют один тип. Общепринятый универсальный способ, в том числе в стандартной библиотеке C++, это передать функцию

два итератора. Например, алгоритм `sort` принимает два итератора, которые обозначают начало и конец диапазона объектов, которые надо сортировать.

```
template<typename RandomIt>
void sort(RandomIt begin, RandomIt end);
```

В чём же плюсы такого подхода? Конечно, универсальность, которую нам здесь дарют шаблоны функций. Вы можете передать итераторы произвольного типа и на элементы произвольного типа, что очень актуально для универсальных алгоритмов типа `sort`

Соответственно, универсальность — это плюс, а в чём же минус? Смотря на сигнатуру функции, вы не можете сходу предположить, особенно если там будут непонятные названия аргументов, например, вы не сможете предположить: функция принимает объекты произвольного или конкретного типа. Эта непрозрачность безусловно является минусом.

В каком же случае можно сделать себе послабления и вместо двух итераторов передать в функцию какой-то конкретный контейнер? Например, если вы от этого контейнера чего-то конкретного ожидаете, например, вы передайте функцию `unordered_set` только потому что вы хотите искать в этом наборе объектов за константное время. Или если вам не нужна такая универсальность, которую вам дают итераторы, если вы хотите максимальной понятности сигнатуры.

Что же делать с константностью? Если вам при вызове этой функции важно, чтобы она не меняла ваши объекты, передайте константные итераторы.

Ещё один интересный момент заключается в том, что вы можете в документации увидеть случаи, когда набор объектов передается вот в таком виде, вот, например,

```
basic_string(const CharT* s,
             size_type count,
             const Allocator& alloc = Allocator());
```

Если вы знакомы с языком C или просто проявляли некоторую любопытность в процессе предыдущих курсов, то вы знаете, что это способ передать набор символов, которые лежат в памяти подряд. В виде указателя на начало этого набора и в виде количества символов в этом наборе. Такое можно встретить не только в конструкторе строки, но и, например, в конструкторе `string_view`.

В чем же проблемы такой сигнатуры функции? В том что человек не особо близко знакомый с C++ или с C, откуда это вообще пришло, может не понять, указатель и `count` каким-то образом другом с другом связаны, что это вообще количество объектов, если отсчитывать от этого указателя столько-то элементов.

Вы уже привыкли, что если вы хотите вызвать какой-то алгоритм от набора элементов, как правило от контейнера, вы должны вызвать `begin`, должны вызвать `end` и вызвать, собственно, нужную вам функцию от этих двух итераторов. Конечно, хотелось бы как в других языках писать просто и компактно, например `sort` от контейнера, и такое скоро можно будет делать. А именно, в ближайшем стандарте C++ скорее всего появится так называемый модуль `Ranges`, который позволит вам не только вызывать алгоритмы от конкретного контейнера, не вызывая `begin` и `end`, но и даже передавать, например, в функцию `sort` ту функцию, которую надо применить к объектам перед их сравнением.

2.3 Как вернуть объект из функции

Настала пора обсудить, как же объекты из функции возвращать. Хотя постойте, мы же это обсуждали в первом курсе. Объекты из функции возвращаются через `return`. Ну то есть здесь ничему новому вы не научитесь...

Ладно, давайте договоримся, что вы будете использовать `return` и только `return`, а мы рассмотрим преимущества этого подхода.

Итак, `return` — это всем известный, максимально удобный и понятный способ вернуть данные из функции. Потому что

- Во-первых, прямо по сигнатуре функции видно, что она возвращает.
- Внутри функции видно выражение `return` что-то, функцию завершили, данные вернули.
- И наконец, специально для `return` придуманы оптимизации `copy elision`, `Named Return Value Optimization (NRVO)`, и, собственно, `move`-семантика, которую мы обсуждали в предыдущем курсе.

Именно поэтому вам стоит максимально использовать `return`.

Однако, конечно, здесь не без подводных камней, и есть ситуации, когда при возврате некоторых объектов из функции у вас копирование неизбежно. А именно, если у объекта много данных на стеке. Давайте рассмотрим такой пример.

```
...
using namespace std;

struct UserInfo {
    string Name;
    uint8_t age;
};

UserInfo ReadUserInfo(istream& in_stream) {
    UserInfo info;
    in_stream >> info.name;
    in_stream >> info.age;
    //...
    return Info;
}

int main() {
    return 0;
}
```

Если вы из функции будете возвращать конкретную переменную, то все тоже будет в порядке, у вас сработает `NRVO`.

Но если вдруг у нас функция будет устроена по-другому

```
...
UserInfo ReadUserInfo(istream& in_stream) {
    /*UserInfo info;
    in_stream >> info.name;
    in_stream >> info.age;*/
    //...
    return ReadUser().info;
}
```

то есть будет возвращаться не локальная переменная, а какой-то другой, пусть даже временный объект, точнее, поле временного объекта, то в этом случае не сработает ни `copy ellision`, ни `NRVO`, а `move`-семантика нам не поможет, потому что у этой структуры со временем будет всё больше и больше данных на стеке. Что же делать в этой ситуации?

Мы могли бы сказать, что нужно отказаться от `return`, но нет. Потому что если вы имеете дело с такими объектами, объектами, у которых много данных на стеке, то у вас будет от них гораздо больше проблем, чем просто при возврате из функции. Например,

```
int main() {
    vector<UserInfo> users;
    sort(begin(users), end(users));
}
```

Выглядит хорошо, но на самом деле сортировка внутри будет переставлять элементы местами, а у этих элементов много данных на стеке, и поэтому это будет долго. Объекты, у которых много данных на стеке, опасны в любом случае. Вам все равно рано или поздно придется переписывать очень много кода, избавляясь от копирований этих объектов. Поэтому есть такой интересный совет по работе с такими объектами, а именно — оборачивать их в `unique_pointer`.

```
...
#include <memory>
...

using InfoPtr = unique_ptr<UserInfo>;

InfoPtr ReadUserInfo(istream& in_stream) {
    const auto info_ptr = make_unique<UserInfo>();
    UserInfo& info = *info_ptr;
    in_stream >> info.name;
    in_stream >> info.age;
    //...
    return info_ptr;
}

int main() {
    vector<InfoPtr> users;
    sort(begin(users), end(users));
}
```

Преимущество этого подхода в том, что мы отселяем объект, у которого много данных на стеке, в кучу. И, благодаря этому, у нас всё становится хорошо. Объекты хранятся в одном конкретном месте и никуда оттуда не перемещаются, только если мы явно не захотим как-то их скопировать.

И теперь у нас и сортировка будет в порядке, потому что будем переставлять указатели на кучу, и вот такие сценарии вида: когда мы хотим оставить каких-то пользователей из диапазона, будут максимально простыми.

2.4 Как вернуть несколько объектов из функции

Что же делать, если вы хотите вернуть из функции несколько объектов, возможно разных типов? На самом деле мы это уже обсуждали в начале второго курса, и на эту тему даже была задача «Трекер задач». В этой задаче нужно было реализовать, в том числе, среди прочих, метод `PerformPersonTasks`, который возвращает два словаря задач. Это мы делали с помощью кортежа.

```
tuple<TaskInfo, TaskInfo> TeamTasks::PerfomPersonTasks(...) {
    ...
    return {move(updated_tasks), move(untouched_tasks)};
}
```

Почему это удобно? Потому что мы могли при вызове этой функцией использовать structured bindings:

```
auto [updated_tasks, untouched_tasks] = tasks.PerformPersonTasks(...);
```

Именно structured bindings позволяют нам здесь использовать возврат с помощью кортежа из нескольких объектов, причем, что ещё хорошо, что даже до structured bindings, если у вас эти переменные уже есть, они как-то объявлены, даже когда не было structured bindings мы могли использовать функцию `tie`:

```
tie(updated_tasks, untouched_tasks) = tasks.PerformPersonTasks(...);
```

Но как правило, конечно, вы используете structured bindings, потому что этих переменных у вас ещё нет.

Это все хорошо и замечательно, но давайте посмотрим на этот код с точки зрения его понятности. Если вы возвращаете из функции наборы объектов совершенно разных типов, и по ним совершенно понятно, какой из них что означает, например:

```
pair<int, CurrencyType> ComputeCost(...);
```

Вот к такой сигнатуре функции вопросов не возникает.

Если же у нас `PerformPersonTasks` возвращает 2 `TaskInfo`, то в принципе, из контекста, не очень понятно, чем они вообще отличаются. И для этих случаев удобно вместо пар и кортежей с непонятными названиями полей, использовать структуры.

```
struct PerformResult {
    TaskInfo updated_tasks;
    TaskInfo untoucned_tasks;
};
```

```
PerformResult TeamTasks::PerfomPersonTasks(...) {
    ...
    return {move(updated_tasks), move(untouched_tasks)};
}
```

У нас никак не меняется `return`-выражение, и, что самое приятное в structured binding, что они умеют распаковывать и структуру тоже.

Хорошо, что ещё можно сказать про возврат нескольких объектов из функции? Не боитесь ли вы, что в `return` выражениях у вас случатся копирования? Конечно, в `return` всё будет хорошо, но здесь есть маленький нюанс, который был виден в коде: если при возврате одной переменной из функций вас спасал `NRVO`, то когда вы возвращаете набор переменных, объединяя их в фигурные скобки, чтобы у вас получилась пара или кортеж, никакого `NRVO` не будет, и вам надо явно вызвать `move` от этих переменных при передаче в конструктор пары или кортежа. Это не очень удобно, но плюсы возврата через `return` и затем связки с помощью structured bindings этот минус перевешивают.

2.5 Возврат данных через исключения

Некоторые из вас могли бы задаться вопросом, почему бы для возврата из функции некоторой дополнительной информации просто не использовать исключения? Что ж, давайте мы на этот вопрос ответим, но с точки зрения производительности.

```
#include "profile.h"
#include <variant>
#include <vector>

using namespace std;

enum class FailReason {
    Bad, Invalid, Ugly, Buggy, Wrong
};

variant<int, FailReason> ComputeCostVariant(int i) {
    if (i % 10 > 0) {
        return i * 100;
    } else {
        return static_cast<FailReason>(i / 10 % 5);
    }
}

int ComputeCostThrow(int i) {
    if (i % 10 > 0) {
        return i * 100;
    } else {
        throw static_cast<FailReason>(i / 10 % 5);
```

```
    }
}

const int ITER_COUNT = 1000000;

int main() {
    vector<int> stats(6);

    {
        LOG_DURATION("variant");
        for (int i = 0; i < ITER_COUNT; ++i) {
            if (const auto res = ComputeCostVariant(i); holds_alternative<FailReason>(res)) {
                ++stats[static_cast<size_t>(get<FailReason>(res))];
            } else {
                stats.back() += get<int>(res);
            }
        }
    }

    {
        LOG_DURATION("throw");
        for (int i = 0; i < ITER_COUNT; ++i) {
            try {
                stats.back() += ComputeCostThrow(i);
            } catch(const FailReason reason) {
                ++stats[static_cast<size_t>(reason)];
            }
        }
    }
    return 0;
}
```

Код скомпилировался, запустился, и мы видим, что версия с `variant` работает быстрее, чем версия с исключениями. То есть версия с исключениями чуточку лаконичнее, но при этом дольше. Какой из этого можно сделать вывод?

Вывод очень простой: **исключения в ваших программах нужны только для действительно исключительных, неожиданных ситуациях или просто не в «горячих» с точки зрения производительности местах**. Когда вы хотите где-то использовать исключения, вспомните о том, не стоит ли вам подумать о производительности и использовать `variant` или `optional`.

Вызовы конструкторов

3.1 Понятность вызовов конструкторов

Давайте обсудим, как же делать вызовы конструкторов более понятными. Вернёмся к задаче «Электронная книга». Давайте представим, что `MAX_USER_COUNT` перестало быть константой и теперь стало параметром конструктора, а также количество страниц у нас тоже как-то ограничено, и это ограничение нам тоже приходит в конструкторе. И давайте представим, что у нас есть некоторый параметр, который как-то меняет рейтинг, как-то на него влияет.

```
class ReadingManager {
public:
    ReadingManager(int max_user_count,
                   int max_page_count,
                   double cheer_factor) : user_page_counts_(max_user_count + 1, -1),
                       page_achieved_by_count_(max_page_count + 1, 0) {}

    void Read(int user_id, int page_count) {
        UpdatePageRange(user_page_counts_[user_id] + 1, page_count + 1);
        user_page_counts_[user_id] = page_count;
    }

    double Cheer(int user_id) const {
        const int pages_count = user_page_counts_[user_id];
        if (pages_count == -1) {
            return 0;
        }
        const int user_count = GetUserCount();
        if (user_count == 1) {
            return 1;
        }
        return (user_count - page_achieved_by_count_[pages_count]) * 1.0
            / (user_count - 1);
    }
private:
    vector<int> user_page_counts_;

    vector<int> page_achieved_by_count_;
```

```
    int GetUserCount() const {
        return page_achieved_by_count_[0];
    }

    void UpdatePageRange(int lhs, int rhs) {
        for (int i = lhs; i < rhs; ++i) {
            ++page_achieved_by_count_[i];
        }
    }
};
```

Как нам создать объект такого класса?

```
ReadingManager manager(20000, 1000, 2);
```

Здравствуй криптографичное создание класса, ничего не понятно — 20000, 1000 и 2. А если поменять их местами, изменится что-то или нет? В общем, непонятно.

Как эту проблему решать? Самый простой способ — это комментарии.

```
ReadingManager manager(/*max_user_count*/ 20000,
                       /*max_page_count*/ 1000,
                       /*cheer_factor*/ 2);
```

На самом деле, уже понятно, что нам не хватает именованных аргументов функции, которые есть в других языках, а в C++ пока, к сожалению, нет, поэтому приходится как-то выкручиваться. И комментарии — это один из способов сделать такой вызов конструктора более понятным. Но и тут есть нюансы. Код стал понятнее, но проблемы остаются.

3.2 Как рефакторить конструкторы с непонятными сигнатурами

Продолжим обсуждать конструкцию со сложными сигнатурами. Мы уже рассмотрели способ комментирования отдельных аргументов конструктора. Давайте научимся это делать более безопасно и прозрачно для компилятора. А именно, если нам не хватало именованных полей, именованных аргументов функций, то придется обходиться существующими средствами языка, и как вариант, можно завести дополнительные методы в этом классе, которые будут устанавливать нужные нам параметры.

```
class ReadingManager {
public:
    ReadingManager() {}

    void SetMaxUserCount(int max_user_count) {
        user_page_counts_.assign(max_user_count + 1, 0);
        user_positions_.assign(max_user_count + 1, 0);
    }
```

```
    void SetMaxPageCount(int max_page_count) {}

    void SetCheerFactor(double cheer_factor) {}
    ...
};
...

int main() {
    ReadingManager manager;
    manager.SetMaxUserCount(20000);
    manager.SetMaxPageCount(1000);
    manager.SetCheerFactor(2);
}
```

Однако есть нюансы: если мы допускаем, что наши старые методы `Read` и `Cheer` могут быть вызваны в том числе и до `Set`-методов, то везде придется добавлять проверки следующего вида

```
class ReadingManager {
public:
    ...
    void Read(int user_id, int page_count) {
        if (max_user_count_ <= 0) {
            // ...
        }
        ...
    }

    double Cheer(int user_id) {
        if (max_user_count_ <= 0) {
            // ...
        }
        ...
    }
    ...
private:
    int max_user_count_ = 0;
    ...
};
```

Выглядит не очень удобно, поэтому давайте думать, что же нам делать с этим классом. Получается, что у его жизненного цикла сейчас есть две явные стадии, которые хорошо видно на примере функции `main`. Мы сначала этот класс инициализируем, с помощью `Set`-методов, а затем используем, и эти стадии хотелось бы явно как-то разграничить, и способ для этого есть: давайте `Set`-методы выделим в отдельный класс. Этот класс мы назовем `Builder`-классом.

```
class ReadingManagerBuilder {
public:
    void SetMaxUserCount(int max_user_count) {
```

```
        max_user_count_ = max_user_count;
    }

    void SetMaxPageCount(int max_page_count) {
        max_page_count_ = max_page_count;
    }

    void SetCheerFactor(double cheer_factor) {
        cheer_factor_ = cheer_factor;
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

Также нам нужно будет как-то обозначить базовый переход из `Builder`-класса в наш класс, в `ReadingManager`, для этого мы напишем метод `Build`, который будет возвращать `ReadingManager`, принимать ничего и будет константным, и в этом методе мы создадим как раз объект `ReadingManager`.

```
class ReadingManager {
public:
    ReadingManager(int max_user_count, int max_page_count, double cheer_factor) :
        user_page_counts_(max_user_count + 1, 0) {

    }

};

class ReadingManagerBuilder {
public:
    ...
    ReadingManager Build() const {
        return {max_user_count_, max_page_count_, cheer_factor_};
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

Получается, что класс нужно создавать теперь следующим образом

```
int main() {
    ReadingManagerBuilder builder;
    builder.SetMaxUserCount(20000);
```

```
    builder.SetMaxPageCount(1000);
    builder.SetCheerFactor(2);
    ReadingManager manager = builder.Build();
}
```

Однако, возникает резонный вопрос, казалось бы нас огорчил вот такой конструктор, с кучей непонятных полей, его всё еще можно вызвать непонятно и печалиться. С этим надо что-то сделать. И еще проверки у нас остались в методах. Проверки можно унести в метод `Build` — и это первое очевидное преимущество

```
    ...
    ReadingManager Build() const {
        if (max_user_count_ <= 0) {
            // ...
        }
        return {max_user_count_, max_page_count_, cheer_factor_};
    }
```

Что же делать с конструктором? Раз уж он такой непонятный и у нас теперь есть способ этот конструктор не вызывать, его можно сделать приватным, то есть не дать его вызывать снаружи.

```
class ReadingManager {
public:
    ...

private:
    ReadingManager(int max_user_count, int max_page_count, double cheer_factor) :
        user_page_counts_(max_user_count + 1, 0) {}
    ...
};
```

Код не компилируется, потому что компилятор ругается на приватный конструктор. Действительно, конструктор приватный, поэтому его вызвать в методе `Build` нельзя. Что же в этом случае делать? Как спрятать тот конструктор от внешних пользователей, но разрешить его вызывать классу `ReadingManagerBuilder`.

Для этого этим двум классам нужно подружиться, а именно нужно в классе `ReadingManager` добавить `friend`**-декларацию**, то есть **класс** `ReadingManager` **будет разрешать методам класса** `ReadingManagerBuilder` **обращаться к своим приватным методам**.

```
class ReadingManager {
public:
    friend class ReadingManagerBuilder;
    ...
```

Попробуем снова скомпилировать код. И он успешно скомпилировался.

Запись в функции `main` можно сделать проще, потому что здесь слишком много раз употребляется этот самый `Builder`, поэтому давайте сделаем такую интересную вещь: мы будем из всех этих `Set`-методов возвращать не `void`, а ссылку на текущий объект `Builder`, неконстантную ссылку.

```
class ReadingManagerBuilder {
public:
    ReadingManagerBuilder& SetMaxUserCount(int max_user_count) {
        max_user_count_ = max_user_count;
        return *this;
    }

    ReadingManagerBuilder& SetMaxPageCount(int max_page_count) {
        max_page_count_ = max_page_count;
        return *this;
    }

    ReadingManagerBuilder& SetCheerFactor(double cheer_factor) {
        cheer_factor_ = cheer_factor;
        return *this;
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

И благодаря этому можно объединять вызовы `Set`-методов в цепочки, не указывая несколько раз название переменной.

```
int main() {
    ReadingManager manager = ReadingManagerBuilder()
        .SetMaxUserCount(20000)
        .SetMaxPageCount(1000)
        .SetCheerFactor(2)
        .Build();
}
```

Проверим, что код компилируется. Код действительно, компилируется.

Таким образом, по сути мы рассмотрели некоторую вариацию **паттерна проектирования** `Builder`, который позволяет нам разграничить инициализацию класса и его последующее использование.

Итак, когда у конструктора, да и вообще у любой функции много параметров, во-первых, рассеивается внимание, во-вторых, очень легко их местами перепутать, поэтому рекомендуется в случае, когда у вас действительно много параметров и это важно для вашего проекта (конечно не во всех случаях) использовать паттерн `Builder`, который позволяет вам, создавая объект типа `Builder` и вызывая у него `Set`-методы и затем метод `Build`, более понятно и прозрачно создавать объекты нужного вам типа.