Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad f*orma*t ting.

МФТИ, ФПМИ, 1 курс, сезон 2018-2019.
Кафедра алгоритмов и технологий программирования
Объектно-ориентированное программирование (язык C++)
Конспект ответов

Берегите наш язык, наш прекрасный язык C++ – это клад, это достояние, переданное нам нашими предшественниками! Обращайтесь почтительно с этим могущественным орудием; в руках умелых оно в состоянии совершать чудеса.
(И. С. Тургенев)

Нам дан во владение самый богатый, меткий, могучий и поистине волшебный язык C++.
(К. Г. Паустовский)

Что язык C++ – один из богатейших языков в мире, в этом нет никакого сомнения.
(В. Г. Белинский)

Язык C++ неисчерпаемо богат и все обогащается с быстротой поражающей.
(М. Горький)

И я надеюсь — мы победим кафедру АТП. Больше: я уверен — мы победим. Потому что разум должен победить.
(Е. Замятин)

Правила экзамена

Экзамен будет состоять из двух частей: предварительной части и основной части.
Предварительная часть проходит по следующим правилам. Экзаменатор (или два экзаменатора) задают вам на свой выбор 4 вопроса, подразумевающих краткие ответы без подготовки (время на раздумье не более одной минуты). Вопросы выбираются из списка "Теорминимум", приведенного ниже. Ответы оцениваются бинарно. Чтобы успешно пройти предварительную часть, необходимо правильно ответить не менее чем на 3 вопроса из заданных четырех. Если вы успешно проходите предварительную часть, то переходите к основной части; иначе уходите с экзамена с оценкой неуд(2).
Гарантируется, что на предварительной части не будут задаваться вопросы, выходящие за рамки списка "Теорминимум" (с той оговоркой, что вопрос вида "Что такое X" всегда подразумевает умение привести пример). Гарантируется, что один вопрос на предварительной части будет покрывать не более одного пункта данного списка (но необязательно будет совпадать с пунктом списка дословно).
Примеры возможных вопросов предварительной части и правильных ответов на них:

- Что такое виртуальная функция? (Это метод класса, который можно переопределить в классах-наследниках так, что реализация метода для вызова будет выбираться во время исполнения, а не во время компиляции.)
- Для решения каких проблем нужна move-семантика? (Например, таких: неэффективный swap двух произвольных объектов, неэффективный push_back в вектор - с демонстрацией на примере кода.)
- Каково действие оператора “запятая”? (Вычислить левый операнд, строго после этого вычислить правый операнд и вернуть результат вычисления правого операнда.)

Основная часть проходит по следующим правилам. Вы тянете билет. Билет будет содержать два вопроса из нижеприведенного списка “Основные вопросы”. Список разбит на две части. Один вопрос в каждом билете будет из первой части, один вопрос - из второй части.
Во время подготовки ответа на билет **можно** пользоваться своими заранее подготовленными рукописными записями в пределах одного листа А4. Пользоваться записями, сделанными на чем-либо, кроме одного заранее выбранного листа А4, или любой печатной (а не рукописной) информацией, или интернет-ресурсами запрещено. Нарушение этого правила приравнивается к списыванию и ведет к удалению с экзамена. (Смысл данного разрешения в том, чтобы каждый мог записать для себя наиболее трудные для запоминания детали. Не следует воспринимать это как призыв постараться законспектировать материал всего курса на одном листе.) С момента начала ответа на билет пользоваться вышеназванным листом запрещается (но, разумеется, можно пользоваться записями, сделанными на другой бумаге во время подготовки ответа на билет).
В ходе основной части экзамена список “Теорминимум” служит ориентиром на случай, если речь идет об оценке уд(3) и ниже. Если в любой момент экзамена выяснится, что Вы не можете ответить хотя бы на несколько вопросов из списка “Теорминимум”, то, вероятнее всего, Вашей оценкой за экзамен будет неуд(2).
Если вы претендуете на оценку выше отл(8), то, скорее всего, вам будут заданы дополнительные вопросы в рамках общей программы курса, приведенной выше (а не только в рамках списков “Основные вопросы” и “Теорминимум”), за исключением того, что выделено красным цветом. В любом случае, экзаменатор, если сочтет нужным, может дополнительно к билету задавать вопросы из любого из трех списков в любом количестве по своему усмотрению.

ТЕОРМИНИМУМ

1. (done) Объясните понятия compilation error, runtime error и undefined behaviour. Приведите по паре примеров того, другого и третьего.

Compile time error aka CE:
Ошибка времени компиляции возникает, когда код написан некорректно с точки зрения языка. Из такого кода не получается создать исполняемый файл.
Примеры:
1) 24abracadabra — лексическая ошибка
2) int const = 5; — синтаксическая ошибка
3) foo(3), хотя сигнатура foo — void foo(int a, int b) — семантическая ошибка
(классификацию ошибок с точки зрения формальных языков, конечно же, знать не нужно)

Runtime error aka RE:

Программа компилируется корректно, но в ходе выполнения она делает что-то непотребное (смотри примеры). RE невозможно отследить на этапе компиляции (компилятор может разве что кинуть предупреждение в месте потенциальной ошибки)
Примеры:
1) Слишком большая глубина рекурсии — получаем stack overflow
2) При попытке записи в невыделенную память операционная система аварийно завершает программу — получаем segmentation fault
3) Деление на ноль
4) Исключение, которое никто не поймал - RE[1]

Undefined behaviour aka UB:
UB возникает при выполнении кода, результат исполнения которого не описан в стандарте. В случае UB компилятор волен сделать, всё что угодно, поэтому результат зависит от того, чем и с какими настройками код был скомпилирован (в теории компилятор может взорвать компьютер). UB может переродиться в CE, RE, или пройти незамеченным и нормально отработать.
Примеры:
1) Для static_cast преобразование указателя на родительский класс к указателю на дочерний класс. Объект по указателю обязан быть правильного дочернего класса, иначе это undefined behaviour.
2) Битые ссылки (подробности в Ч1, 1).
3) ++x = x++; или f(x=y, x=3); — порядок вычисления аргументов оператора и функций не определён. (until C++17)
4) int x = 2 << 40; — не определено, что будет происходить при переполнении *знакового* типа.
5) Чтение выделенной, но неинициализированной памяти. В теории, считается какой-то мусор, но технически, так как это UB, компилятор в праве поджечь ваш монитор.
6) Отсутствие return в конце функции, которая что-то возвращает. Шок, да? Это не ошибка компиляции! Это — чёртово UB! Зачем?!
7) Выход за границы C-style массива.

Дополнительно:
Зачем в стандарте оставлять такие дыры, спросите вы? Не лучше бы в стандарте описать вообще всё? Так же будет куда меньше непонятных ошибок! Хорошие пояснения со *стекперелился*: https://stackoverflow.com/a/36524349 и https://stackoverflow.com/a/51558016

2. (done) Что такое выражения, операторы? Для чего предназначены и что по стандарту делают следующие операторы: тернарный оператор, оператор “запятая”, унарная звездочка, унарный амперсанд, операторы “точка” и “стрелочка”, двойное двоеточие, префиксный и постфиксный инкремент, бинарные & и &&, операторы простого и составного присваивания, операторы << и >>?

1) Идентификаторы — любая последовательность латинских букв, цифр и знака “_”, не начинающаяся с цифры. Они не могут совпадать с ключевыми словами (`new`, `delete`, `class`, `int`, `if`, `true`, etc)
2) Литералы — последовательность символов, интерпретируемая как константное значение какого-то типа (`1`, `'a'`, `"abc"`, `0.5`, `true`, `nullptr`, etc)
3) Операторы — это, можно сказать, функции со специальными именами (`=`, `+`, `<`, `[]`, `()`, etc)
4) Выражение — некоторая синтаксически верная комбинация литералов и идентификаторов, соединенных операторами
5) Тернарный оператор (`?:`). “Условие” ? “выражение, если true” : “выражение, если false”
6) Оператор “запятая” (`,`). Вычисляет то, что слева, *затем* вычисляет то, что справа и возвращает то, что справа. (Имеет самый низкий приоритет)
7) Унарная “звёздочка” (`*`). Разыменование
8) Унарный “амперсанд” (`&`). Взятие адреса

[1] Note: не всякое исключение есть RE, и не каждое RE есть исключение.

9) Оператор “точка”/”стрелочка”. Доступ к полю/методу класса (соответственно через объект класса/указатель на объект)
10) Двойное двоеточие. Переход в другую область видимости (`std::cout`, `::operator new`)
11) Префиксный/постфиксный инкремент. Префиксный увеличивает на единицу и возвращает ссылку на уже измененный объект. Постфиксный увеличивает на единицу и возвращает копию старого объекта.
12) Бинарный амперсанд. Побитовое И
13) Бинарный двойной амперсанд. Логическое И
14) Оператор присваивания. Присваивает значение (копированием или перемещением)
15) Оператор составного присваивания. Легче пример: a += 5 ~ a = a + 5. Только во втором случае создается лишняя копия
16) Оператор << (>>). Ну, либо побитовое смещение влево (вправо), либо это оператор вывода(ввода) из(в) поток(а).

3. (done) Что такое стековая память и динамическая память? Что такое stack overflow? Зачем нужен оператор new? Что такое утечки памяти? Что такое сборка мусора?

1) Стековая[2] память — память, в которой находятся все локальные переменные функций. Её у нас условно где-то несколько мегабайт
2) Динамическая память — некоторая память процесса, которая выдаётся нам по запросу (через new/malloc/calloc). Её гораздо больше, чем стековой (столько, сколько есть у всей системы). Обращение к ней происходит через указатели
3) Stack overflow — ~~источник всех знаний в этой вселенной~~ переполнение стековой памяти. (Слишком глубокая рекурсия или обычный массив большого размера). Это RE
4) Утечка памяти — это когда вы выделили динамическую память, но не вернули её системе по окончании её использования (или при каком-то аварийном завершении).
5) Сборка мусора — механизм автоматического управления памяти, когда ты не заботишься о том, чтобы вызвать delete, система всё делает за тебя. Есть в Джаве (Java) и Питоне (Python), в плюсах (C++) отсутствует (хотя умные указатели — вполне неплохая альтернатива)

4. (done) Объясните идею ссылок (references). Для чего они нужны, чем они отличаются от указателей? Что такое “передача аргументов по ссылке и по значению”? Как в C++03 реализовать функцию swap?

Ссылки используются как псевдонимы для переменных. Отличия от указателей не только синтаксические, но и семантические:

- ссылку необходимо инициализировать в момент создания;
- нельзя “перепривязать” ссылку к другому объекту;
- не имеют адреса;
- не могут быть элементами массива;
- существует арифметика указателей, но нет арифметики ссылок.

Ссылки могут быть использованы в аргументах функций. Передача аргументов по ссылке не создаёт копии аргументов, в то время как передача по значению копирует аргументы.

Реализация swap (шаблоны см. в 9):

```
template <typename T>
void swap(T& a, T& b) { // a и b не будут скопированы
 T c = a;
```

[2] Можно спутать со статической, но судя по всему, это не одно и то же: “Статическая память может трактоваться ещё и как память, в которой лежат глобальные переменные, а они, очевидно, лежат не на стеке.“ — прим. тов. Гусева

```
 a = b;
 b = c;
}
```

5. (review) Что такое класс, структура, в чем разница между ними? Что такое поля, методы, модификаторы доступа, инкапсуляция?

1) Класс/структура — пользовательский тип данных. Отличий 2. Первое: по умолчанию все поля/методы в классе приватные, в структуре — публичные. Второе: по умолчанию наследование от чего-то, когда ты класс — приватное, когда ты структура — публичное.
2) Поля — переменные, связанные с классом. Все данные класса хранятся в его полях (ну, за исключением мб таблицы виртуальных функций ;)
3) Методы — функции, определенные внутри класса.
4) Модификаторы доступа — `public`, `protected`, `private` (ключевые слова, служащие для сокрытия данных). `public` — поле/метод доступны снаружи класса, `protected` — поле/метод доступны только наследникам, `private` — поле/метод доступны только внутри класса.
   (Однако если объявить класс B *другом* класса A (написать `friend class B;` в теле A), то класс B получит доступ ко всем членам класса A независимо от уровня доступа к ним. Аналогичное правило действует для *дружественных функций*.)
5) Инкапсуляция — механизм упаковки данных в единый компонент (класс)

6. (done) Что такое конструкторы, деструкторы, для чего они нужны, каков синтаксис их определения? Те же вопросы про конструктор копирования и оператор присваивания.

1) Конструктор — специальный блок инструкций, вызываемый при создании объекта класса
2) Деструктор — при уничтожении объекта (для статического — при выходе из области видимости, для созданного в динамической памяти — при delete) Можно и явно написать a.~Myclass()
3) Конструктор копирования — это тот же конструктор, только от объекта этого же класса (точнее, от const ссылки на этот объект, иначе вы словите stack overflow из-за рекурсивного вызова конструктора копирования в аргументах конструктора)
4) Оператор присваивания копированием — особый оператор, используемый для присваивания объектов одного класса друг другу

```
vector(аргументы) : список инициализации {тело}
vector(const vector& other);
vector& operator=(const vector& other);
~vector() {тело}

vector<int> first;
vector<int> second(first);  // тут вызывается конструктор копирования
vector<int> third = second;  // и тут тоже
first = second;  // А тут - оператор присваивания копированием
```

Примечания:

1. Деструктор не должен принимать аргументов
2. Правило трёх: если ты реализовал какой-то нетривиальный деструктор/конструктор копирования/оператор присваивания копированием, то следует реализовать и остальные два
3. Правило пяти (since C++11): если ты реализовал какой-то нетривиальный деструктор/конструктор копирования/оператор присваивания копированием/конструктор

перемещения/оператор присваивания перемещением, то следует реализовать и остальные четыре
4. Также можно сделать конструктор по умолчанию (= default)
5. Также можно запретить вызывать всё из вышеперечисленного (= delete)

7. (done) Что такое перегрузка функций? Что такое перегрузка операторов? Приведите примеры операторов, которые можно и нельзя перегружать. Приведите пример перегрузки хоть какого-нибудь оператора (напишите сигнатуру его перегрузки).

1) Перегрузка функций — возможность создавать несколько одноименных функций, но с различными параметрами.
2) Перегрузка операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.
3) Можно перегружать: бинарные `+`, `-`, унарную `*`, бинарную `*`, `++`, `+=` и т.д.
4) Можно перегружать только в качестве методов класса: `=`, `()`, `[]`, `->`, операторы приведения типов
5) Нельзя перегружать: `?:` (тернарный), `::`(доступ к области видимости), `.`(точка; доступ к полям), `.*`(точка со звездочкой; доступ к полям по указателю), `sizeof`, `typeid`

   Пример перегрузки оператора `[]`:

```
T& operator[](int index) { return data[index]; }
```

8. (done) Что такое наследование? В чем разница между приватным и публичным наследованием?

1) Наследование — это одна из основных концепций ООП, позволяющая создавать классы на основе других классов, при этом заимствуя их функционал
2) При приватном наследовании только сам наследник внутри себя (и его друзья) знают о факте наследования, то есть имеют доступ к полям/методам родителя. А извне мы не можем через объект класса-наследника обратиться к чему угодно из класса-родителя
3) При публичном мы знаем извне, что данный класс — наследник, и потому можем обращаться к чему угодно, лежащему в классе-родителя (если оно public, конечно же).

9. (review) Что такое виртуальные функции, чисто виртуальные функции, абстрактные классы? Что такое виртуальное наследование?

1) Виртуальные функции — такие методы класса, реализация для которых при обращении к объекту класса-наследника через указатель/ссылку на какого-то из его предков, у которого (или у предка которого) эта функция помечена `virtual`, берется из класса того самого наследника (наследник не обязательно прямой)
2) Чисто виртуальные функции — функции, в сигнатуре которых написано “`= 0`”. Они обязаны быть определены у прямых потомков, иначе они [потомки] сами станут абстрактными классами
3) Абстрактный класс — класс, в котором есть хоть одна чисто виртуальная функция. Объекты таких классов нельзя создать.
4) Виртуальное наследование — наследование, помеченное словом `virtual` (то есть, класс из списка базовых классов может быть помечен этим словом). Допустим, при ромбовидном наследовании, если “отцы” будут виртуально унаследованы от “деда”, то создастся только одна версия “деда”.

10. (done) Что такое полиморфизм? Приведите пример полиморфизма в C++.

1) Полиморфизм — способность функции обрабатывать данные разных типов (т.е. возможность писать универсальный код для объектов разных типов). Бьёрн Страуструп определил полиморфизм как «*один интерфейс — много реализаций*».

2) В C++ есть три проявления полиморфизма: перегрузка функции, виртуальные функции и шаблоны. Разберём на примере шаблонов. Действительно, шаблоны — это статическое проявление полиморфизма, позволяющие объявлять (обобщённо) вещи, универсальные для всех типов. Например, ниже написана функция, которая для любого типа выводит его размер:

```
template <typename T>
void TypeSize(const T& x) {
 std::cout << sizeof(x) << ' ';
}

int main() {
 char x = 97;
 double y = 1.0;
 TypeSize(x), TypeSize(y); // prints 1 8
}
```

11. (done) Что такое шаблоны? Что такое инстанцирование шаблонов, специализация шаблонов? Приведите пример каких-нибудь шаблонов из STL, обладающих специализацией.

1) Шаблоны — это средство языка, предназначенное для написания кода без привязки к конкретному типу.

2) Инстанцирование шаблона — процесс создания конкретного класса/функции/`using` (см. далее) из шаблона путем подстановки аргументов. Процесс инстанцирования шаблонов происходит м/у препроцессингом и компиляцией. Происходит с проверкой типов аргументов (совместимость по присваиванию, по приведению типов, по вызываемым методам). Для классов нужно явное инстанцирование[3].

3) Специализации шаблонов нужны для случаев, когда мы хотим, чтобы с данным набором типов данных функция вела себя по-другому (т.е. как-то специально). Специализации активно используются в type_traits. Вот пример (STL-ной реализации с сайта cppreference):

3.1) `std::remove_reference` “убирает” ссылку у типа, или, например, std::vector имеет специализацию для bool. Код:

```
template<typename T>
struct remove_ref {
   typedef T type;
};

template<typename T>
struct remove_ref<T&> {
   typedef T type;
};

template<typename T>
struct remove_ref<T&&> {
   typedef T type;
};
```

12. (done) Что такое исключения? Как пользоваться механизмом обработки исключений? Как бросить, поймать исключение? Приведите какой-нибудь пример.

1) Исключение — нестандартная ситуация, возникающая в ходе выполнения программы.

---

[3] С C++17 существует автоматический вывод параметров шаблонов, то есть в выражениях вроде std::pair p(12, 23) или std::vector v{12, 23, 43} будет всё норм

2) Механизм генерации исключений в C++ реализован с помощью ключевого слова `throw`, обработки — с помощью `try` и `catch`. Блок кода, который потенциально выкидывает исключение, можно “обернуть” в try-блок, который при возникновении исключения сообщит об этом catch-блоку, *следующему сразу за try-блоком*. Последний, в свою очередь, проверяет исключение на соответствие, и, если нашёлся подходящий обработчик, исключение считается обработанным. Если же ни один из обработчиков не смог обработать исключение, оно пробрасывается дальше на более высокий уровень (т.е. программа выходит из вложенных функций). Если исключение не было обработано на последнем (глобальном) уровне, то программа завершается с RE посредством вызова std::terminate. Пример:

```
int main() {
 // try-блок
 try {
   int a = 5;
   throw a;  // генерация исключения
   a = 6;  // код после throw никогда не будет достигнут
 }
 // catch-блок: начало
 catch (int error) {
   if (error == 0) {
     // do something
   } else if (error == 1) {
     // do something
   } else {
     throw error;
   }
 } catch(char a) {
   // do something
 }
 // catch-блок: конец
}
```

13. (review) Рассмотрим контейнеры std::vector, std::list, std::deque. Каковы основные операции, предоставляемые ими, и скорость работы этих операций?

1) std::vector
    a. operator[] — обращение по индексу — O(1)
    b. push_back — вставляет элемент в конец — амортизированная O(1)
    c. pop_back — удаляет из конца — O(1)
    d. size — возвращает число элементов — O(1)
    e. resize — изменяет количество хранимых элементов — O(n)
    f. reserve — увеличивает размер буфера — O(n)
    g. insert — вставляет эл-ты перед указанной позицией — O(n)
    h. erase — удаляет эл-ты из заданного диапазона — O(n)
2) std::list
    a. insert — добавление элемента в произвольное место (по итератору) — O(1)
    b. erase — удаление по итератору — O(1)
    c. size, push_back, pop_back, push_front, pop_front — аналогично — O(1) [не амортиз.]
    d. front, back — доступ к первому и последнему элементу — O(1)
3) std::deque
    a. push_back, pop_back, push_front, pop_front — амортизированная O(1)
    b. operator[] — доступ по индексу — O(1)

14. (review) Рассмотрим контейнеры std::map, std::set, std::unordered_map, std::unordered_set. Каковы основные операции, предоставляемые ими, и скорость работы этих операций?

За всеми подробностями обращайтесь к вопросам 17 и 18 основной части. Здесь же напоминание по самым основным функциям.

Первые два работают на красно-чёрном дереве[4], последние два — на хеш-таблицах

1) std::map — это отсортированный ассоциативный (т.е. нет понятия первый/последний элемент[5], но есть понятия ключ и значение, которое ему соответствует) контейнер, который содержит пары ключ-значение с неповторяющимися ключами.
   Основные методы std::map:
   1. *begin() (cbegin())* возвращает итератор (const итератор) на первый элемент, O(1)
   2. *end() (cend())* возвращает итератор (const итератор) на элемент, следующий за последним, O(1)
   3. *size()* возвращает количество элементов в контейнере, O(1)
   4. *insert()* вставляет эл-ты, O(log(n)) (для диапазона — O(k*log(n+k)), k — кол-во вставленных)
   5. *erase()* удаляет элементы, O(log(n)) + O(k), где k — либо расстояние (если удаление диапазона), либо число удалённых элементов (если удаление по значению)
   6. *find()* находит элемент с конкретным ключом, O(log(n))
   7. *Также стоит упомянуть,* что *operator[]* коварный, а именно:
      a. Его нельзя использовать с *const map*, так как он неконстантный
      b. если по данному ключу не было элемента, то он создаст элемент по этому ключу (чтобы не кидать исключение), проинициализирует его по умолчанию и вернёт значение по умолчанию
      c. Если же Вы хотите обращение, кидающее исключение вместо создания, то используйте константный метод *at()*
2) std::set — ассоциативный контейнер, который содержит упорядоченный набор уникальных объектов.
   Основные методы: те же, что и у *map*, только нет *operator[]* и *at()*.
3) std::unordered_map — это ассоциативный (т.е. нет понятия первый/последний эл-т, но есть понятия ключ и значение, которое ему соответствует) контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Элементы не отсортированы, но распределены по корзинам (распределение полностью зависит от значения хеша ключа).
   Основные методы std::unordered_map:
   1. *begin() (cbegin())* возвращает итератор (const итератор) на первый элемент, O(1)
   2. *end() (cend())* возвращает итератор (const итератор) на элемент, следующий за последним, O(1)
   3. *size()* возвращает количество элементов в контейнере, O(1)
   4. *insert()* вставляет эл-ты, O(1) в среднем, O(n) — в худшем (если диапазон, то в среднем — O(k), в худшем — O(k*n + k))
   5. *erase()* удаляет элементы, O(k) — в среднем, O(n) — в худшем, где k — либо расстояние (если удаление диапазона), либо число удалённых элементов (если удаление по значению)
   6. *count()*[6] возвращает количество элементов, соответствующих определенному ключу, O(k) — в среднем (k — число найденных элементов, т.е. константа), O(n) — в худшем

---

[4] В стандарте не указано, как именно должны быть реализованы map и set, но обычно в реализациях стандартной библиотеки используется именно RBT

[5] Имеется в виду в смысле порядка расположения эл-тов.

[6] В unordered_map возвращает 1 или 0, в unordered_multimap она может вернуть что-то ещё. В случае не multi, count часто используется как удобная проверка наличия элемента в контейнере

7. *find()* находит элемент с конкретным ключом, O(1) в среднем, O(n) — в худшем
8. Для *unordered_map* стоит отметить, что *operator[]* коварный, а именно:
    a. его нельзя использовать с *const unordered_map*, так как он неконстантный
    b. если по данному ключу не было элемента, то он создаст элемент по этому ключу (чтобы не кидать исключение), проинициализирует его по умолчанию и вернёт значение по умолчанию
    c. Если же Вы хотите обращение, кидающее исключение вместо создания по умолчанию, то используйте константный метод *at()*

4) std::unordered_set — ассоциативный контейнер, который содержит набор уникальных объектов.
   Основные методы: те же, что и у *unordered_map*, только нет *operator[]* и *at()*.

15. (done) Что такое итераторы? Какие виды итераторов существуют? Зачем они вообще нужны? Какие итераторы поддерживает каждый из контейнеров, упомянутых в предыдущих двух вопросах?

Итератор — сущность, позволяющая выполнять обход элементов контейнера. Для итератора должно быть определено понятие “следующий элемент” — т.е. к каждому итератору применим пре-инкремент (`++iter`), и доступ *на чтение* к текущему элементу — т.е. разыменовывание (`*iter`).
Суть итераторов в том, чтобы общим интерфейсом уметь итерироваться по разным контейнерам. Это позволяет писать более общие алгоритмы, которые работают с любыми контейнерами, поддерживающими заявленный вид итераторов.
Существует 5 видов итераторов:

Input
- позволяет читать данные
- пройтись по контейнеру можно один раз (т.е. не гарантируется, что последующие проходы возможны)

Output
- позволяет изменять данные (т.е. для него определено выражение `*iter = value`)
- пройтись по контейнеру можно один раз (см. выше)

Forward — подмножество Input (т.е. он может всё что может Input)
- может быть скопирован и использован для повторного обхода

Bidirectional — подмножество Forward
- может быть уменьшен на единицу (т.е. разрешён обход в обратную сторону)

Random Access — подмножество Bidirectional
- поддерживает арифметические операции `+` и `-` (второй аргумент — число; `+` коммутативен)
- поддерживает сравнения (`<`, `>`, `<=`, `>=`) между итераторами
- поддерживает операции увеличения `+=` и уменьшения `-=`
- поддерживает разыменование по индексу `[]` (конструкция `iter[index]` равносильна `*(iter + index)`

Если итератор удовлетворяет и свойствам Output, и свойствам Input/Forward/Bidirectional/Random Access, то он называется “*mutable* Input/Forward/Bidirectional/Random Access iterator”.

Виды контейнеров и итераторы:
```
forward_list, unordered_* - Forward
list, set, map - Bidirectional
vector, deque, array - Random Access
```

16. (done) Что такое компараторы? Что такое функциональные объекты? Приведите хоть один пример.

Компаратором называется объект, позволяющий сравнивать элементы и удовлетворяющий следующим свойствам:

```
I.  Exactly one of the following statements is true:
    a) comp(x,y) is true  and comp(y,x) is false
    b) comp(x,y) is false and comp(y,x) is true
    c) comp(x,y) is false and comp(y,x) is false
II. If comp(x,y) and comp(y,z) are true, then so is
     comp(x,z) (transitivity).
```

Т.е. оператор должен удовлетворять отношению *строгого линейного порядка*. Подробнее

Компаратором может быть:

- объект класса, реализующего `operator()`, который имеет следующую сигнатуру:
  ```
  bool operator()(const C& x, const C& y) const; // C - класс сравниваемых объектов
  ```
  Такой объект называется функциональным объектом или функтором.
  (лямбда-выражение — тоже функтор!)
- функция с сигнатурой, указанной выше

Пример функтора:

```
struct Cmp {
 bool operator()(const std::string& x, const std::string& y) const {
   return x.size() < y.size();
 }
};

int main() {
 std::vector vector = {"z", "c", "fun", "b", "asd", "ss"};
 Cmp cmp;  // объект компаратора
 // передаём функц. объект в std::sort в качестве оператора сравнения
 std::sort(vector.begin(), vector.end(), cmp);
 // vector: z c b ss fun asd
}
```

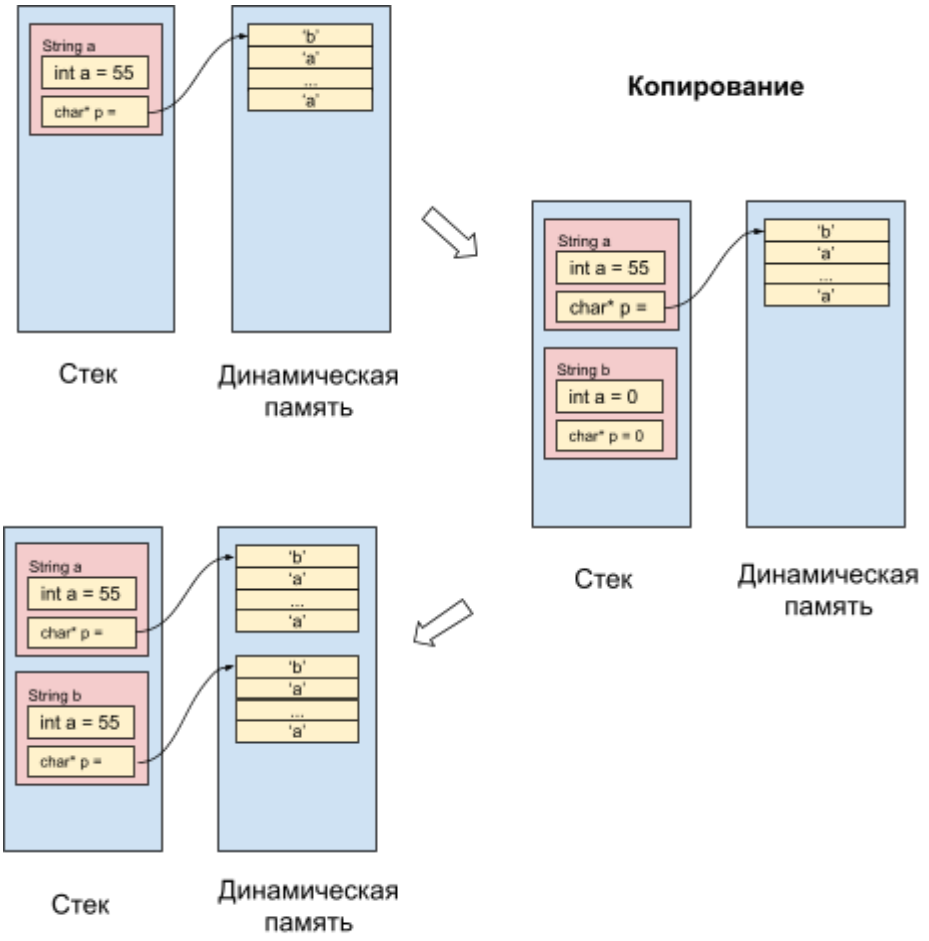
17. (done) Для чего нужна move-семантика? Расскажите в общих чертах, что это такое. Как правильно реализовать функцию `swap` в C++11?

До C++11 был только один способ передачи объекта в другую область памяти — полное копирование (передача по ссылке не считается, так как это выдача другого имени на ту же память), которое очень дорогостоящее. Можно ли как-то схитрить и делать копирование быстрее? Можно, если у нас есть дополнительное условие, что исходный объект после передачи больше не пригодится. Назовём такое хитрое копирование *перемещением*. Оно, как и копирование, происходит в своём специальном конструкторе — *конструкторе перемещения.*

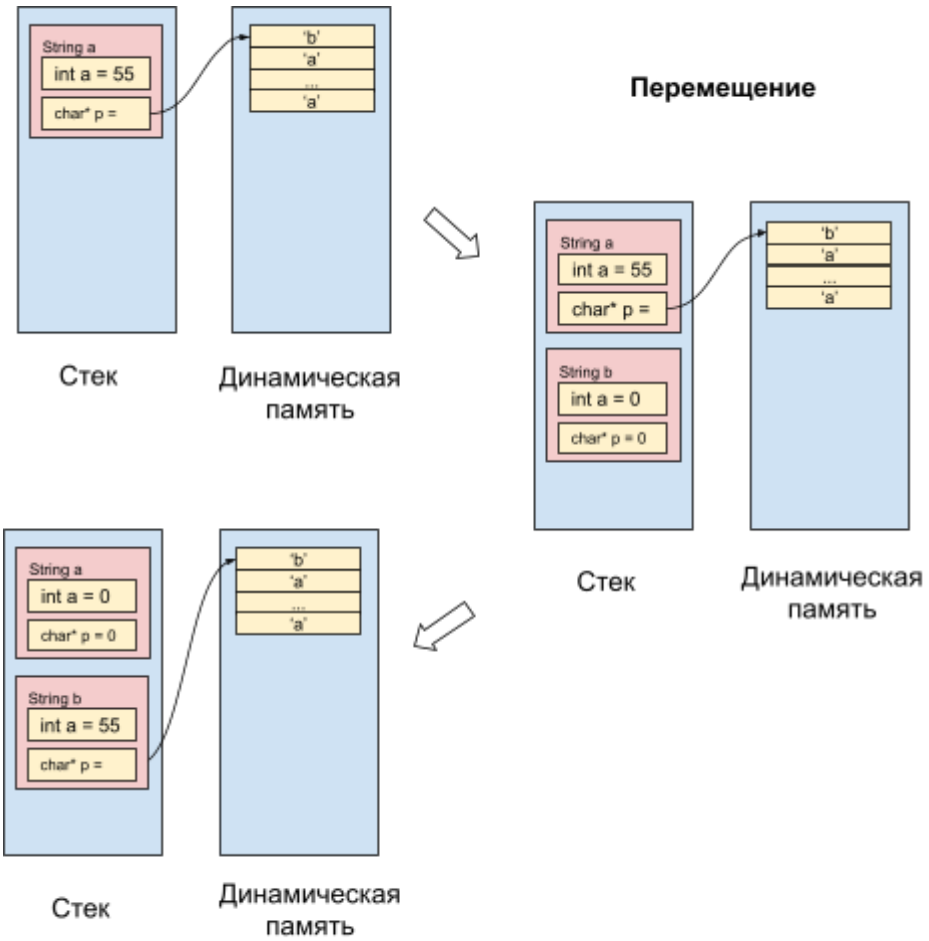
В чём заключается оптимизация при перемещении? Примитивные типы на стеке копируются точно так же, как и при обычном копировании. Идея: данные, хранящиеся в динамической памяти, можно не копировать — можно просто скопировать указатель на эту память. Но ведь тогда произойдёт двойное очищение памяти: при разрушении старого и нового объекта! И вообще, изменение одного объекта будет приводить к изменению другого, и изменения могут не согласовываться! Вспоминаем: у нас есть дополнительное знание — старый экземпляр объекта

больше не будет использоваться. Следовательно, все указатели на динамическую память в исходном объекте можно обнулить (см. рисунок), тем самым сделав объект valid but in unspecified state.

Если объект состоит из сложных типов, то происходит их перемещение, при котором применяется та же логика.


Копирование
String a
int a = 55
char* p =
'b'
'a'
...
'a'
String b
int a = 0
char* p = 0
Стек
Динамическая память

String a
int a = 55
char* p =
String b
int a = 55
char* p =
'b'
'a'
...
'a'
Стек
Динамическая память


Перемещение
String a
int a = 55
char* p =
String b
int a = 0
char* p = 0
'b'
'a'
...
'a'
Стек
Динамическая память

String a
int a = 0
char* p = 0
String b
int a = 55
char* p =
'b'
'a'
...
'a'
Стек
Динамическая память

Примеры соответствующих конструкторов:

```
String(const String &another) {
   a = another.a; //copy trivial types
   p = new char[a]; //allocating new chunk of memory
   memcpy(p, another.p, a); //copies all dynamic memory
}

String(String &&another) noexcept {
   a = another.a; //copy trivial types
   p = another.p; //steal new string
   another.a = 0; //clearing old object
   another.p = nullptr; //clearing old object
}
```

Соответственно, move-семантика была введена для того, чтобы правильно реализовать операцию перемещения. Для перемещения было необходимо ввести новый вид ссылок со следующими свойствами:
1) Возможность привязываться и к временным (мы же не хотим копировать только что созданный объект, мы хотим его перемещать), и к невременным объектам (если старый объект нам не нужен)
2) Возможность изменять старый объект (иначе будет двойное очищение одного участка памяти)

Такие ссылки называются *rvalue-ссылками*. Они работают как обычные ссылки, но смотри выше.

Для того, чтобы привязать rvalue-ссылку к “невременному” объекту (в том числе для передачи объекта в конструктор перемещения, который принимает type&& — в этом, собственно, наша цель и состоит), надо скастовать его к rvalue-ссылке[7] с помощью std::move

```
String s("I hate myself", 13);
String&& ss = s; //does not compute
String&& sss = std::move(s); //compute!
```

В некотором смысле с помощью std::move мы помечаем объект как временный => далее в коде не нужный => его можно переместить => обнулить. Сам std::move, кстати, объект не портит.

Реализация std::swap

```
template <typename T>
void swap(T& a, T& b){
   T t = std::move(a);
   a = std::move(b);
   b = std::move(t);
}
```

18. (review) Что такое умные указатели? Для решения каких проблем они нужны? Приведите пример использования.

Существует три вида умных указателей: unique_ptr, shared_ptr и weak_ptr.
(auto_ptr был удалён в C++17. RIP)

unique_ptr и shared_ptr решают проблему автоматического очищения памяти при выходе указателя из области видимости (так как можно легко потерять delete, соответствующий какому-то new, например, если между new и delete бросится исключение и delete не вызовется, смотри пример далее). Разница unique_ptr и shared_ptr — unique_ptr нельзя копировать, он реализует

[7] Выражение std::move(x) является xvalue, которое, в свою очередь, является подвидом rvalue.

идею единоличного владения объекта, на который он указывает, shared_ptr можно копировать, причём объект под указателем удаляется тогда, когда не останется ни одного экземпляра shared_ptr.

weak_ptr решает проблему проверки, был ли объект под указателем уже удалён или нет (если попытаться разыменовать указатель, объект под которым уже был удалён, получим ошибку; проверить, был ли этот объект удалён с помощью только сырого указателя, нельзя). Также он решает проблему закольцованности ссылок: если объекты указывают друг на друга через shared_ptr, они никогда не будут удалены; в этом случае один из указателей стоит заменить на weak_ptr, так как он не влияет на счётчик shared_ptr.

Примеры использования:

Вообще, по хорошему умные указатели стоит использовать как можно чаще вместо сырых потому что они куда безопаснее. Вот рафинированный пример, когда без shared_ptr не обойтись:

```
void foo() {
   throw "catch me if you dare!";
}

void bar(){
   int* p = new int(5);
   foo();
   //never will be executed => memory leak
   delete p;
   /* ... */
}
```

Если использовать умные указатели:

```
#include <memory>

void foo(){
   throw "catch me if you dare!";
}

int bar(){
   //two ways of constructing shared_ptr
   std::shared_ptr<int> p(new int(5));
   auto pp = std::make_shared<int>(5);
   foo();
   //memory will be cleaned
   /* ... */
}
```

Пример с weak_ptr, тоже рафинированный:

```
int* ptr = new int(10);
int* ref = ptr;
delete ptr;
```

При разыменовывании ref будет ошибка, решение (lock — метод, возвращающий shared_ptr по тому адресу, на который указывает weak_ptr, шок, да?):

```
std::shared_ptr<int> sptr(new int(10));
std::weak_ptr<int> weak1 = sptr;
/* ... */
sptr.reset(new int(5));
/* ... */
std::weak_ptr<int> weak2 = sptr;
if(auto tmp = weak1.lock())
   std::cout << *tmp;
else
   std::cout << "weak1 is expired";
if(auto tmp = weak2.lock())
   std::cout << *tmp;
else
   std::cout << "weak2 is expired";
```

(Пример с зацикливанием ссылок см. в билете, не буду его сюда дублировать)

Хороший пример реального использования (сос мыслом): допустим есть набор объектов, хранимых где-либо, и есть очередь обработки этих объектов, записанная в вектор. В этом векторе будем хранить weak_ptr'ы — тогда, когда объект будет удалён, мы сможем легко это проверить и удалить объект из очереди. Почему бы не хранить shared_ptr? Вот почему: мы не хотим очередью "владеть" этим объектом, ведь очередь всего лишь задаёт порядок обработки, и если настало время удалить объект, а в очереди хранится shared_ptr, то объект не будет удалён.

19. (done) Что такое аллокаторы? Какова общая идея класса std::allocator, как и для чего он используется?

Аллокатор — объект, который выделяет/освобождает память и создаёт/разрушает объекты на выделенной памяти.
Зачем нужна эта прослойка? — можно сделать кастомные аллокаторы, с помощью которых выделение памяти эффективнее для какой-то специфической ситуации.
Стандартные функции аллокатора: `allocate`, `deallocate`, `construct`, `destroy`. Последние два просто вызывают placement-new и деструктор объекта, поэтому они одинаковы для всех аллокаторов.
`std::allocator` — простая обёртка над тривиальными вызовами `operator new` и `operator delete`. Он используется как, например, аллокатор по умолчанию в контейнерах STL.

20. (done) Что такое лямбда-выражения, каков их синтаксис? Приведите хоть один пример использования.

1) Лямбда-функциями (-выражениями) называются безымянные локальные функторы, которые можно создавать прямо внутри какого-либо выражения.
Синтаксис:

```
[список захвата] (аргументы) /*опционально: mutable*/
/*опционально: noexcept*/ /*опционально: "-> тип возвращаемого значения"*/
{ тело функции };
```

Примеры реального использования:
1) Возвращает количество четных элементов вектора.

```
std::count_if(vec.begin(), vec.end(), [] (int n) { return (n % 2) == 0; });
```

2) Умножает элементы вектора на 2.

```
std::for_each(vec.begin(), vec.end(), [] (int& n) { n *= 2; });
```

21. (done) Напишите вычисление n-го числа Фибоначчи (в пределах long long) на этапе компиляции, не используя слова constexpr.
В C++03 это будет выглядеть след. образом (static нужен, чтобы value было именно характеристикой класса, а не объекта класса, чтобы можно было к нему так обращаться):

```
template<int n>
struct SFibonacci {
   static const int value = SFibonacci<n - 1>::value + SFibonacci<n -
2>::value;
};

template<>
struct SFibonacci<1> {
   static const int value = 1;
};

template<>
struct SFibonacci<0> {
   static const int value = 0;
};

int main() {
   const int n = 10;
   std::cout << SFibonacci<n>::value; // prints 55
}
```

22. (done) Что такое SFINAE? Приведите хотя бы один пример (на уровне идей, можно без реализации).
SFINAE *(substitution failure is not an error)* — это правило, которым руководствуется компилятор при создании необходимого кода. Хорошая демонстрация:

```
template <typename T>
typename T::type f(T x) {
 std::cout << 1;
 return x;
}

template <typename...>
int f(...) {
 std::cout << 2;
 return 1;
}

int main() {
 f(5); // prints 2
}
```

Поясним: на этапе выбора перегрузки функции компилятор смотрит все возможные версии, выбирает одну из них в соответствии с правилами. При разрешении перегрузки не рассматривается тип возвращаемого значения, поэтому выбирается версия с только T как более частная. Компилятор должен понимать, что это за функция, поэтому он подставляет T=int, проверяет сигнатуру и тип возвращаемого значения. Но у типа int нет никакого type, происходит ошибка. Тогда компилятор смотрит ещё доступные версии (в нашем случае осталась только одна), поэтому выбирается вариант с произвольным числом параметров.
(Ещё раз отметим, что ошибка должна возникать именно на этапе перегрузки функции, а не при выборе специализации шаблонов или в теле функции, иначе SFINAE не работает).

ОСНОВНЫЕ ВОПРОСЫ

ЧАСТЬ 1

1. (done) Расскажите об указателях и ссылках в C++03. В чем их идея, для чего они нужны? В чем разница между указателями и ссылками? Какие операции они поддерживают, чем их можно, а чем нельзя инициализировать? Что такое константные указатели, указатели на константу, константные ссылки, в чем проявляется разница между ними и обычными указателями и ссылками? Расскажите о проблеме “битых ссылок”, приведите пример.

1) Указатель — переменная, значением которой является адрес ячейки памяти.
Ссылка — особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Ссылка — это новое название для уже существующей переменной.
2) Отличие ссылок от указателей (оба ответа). Из этих свойств вытекает:
C++ Standard §8.3.2/4:
There shall be

- no references to references
  T.е. недопустим код:
  ```
  int a = 0;
  int& & b = a;
  //  ^---- пробел обязателен, т.к. слитное написание означает rvalue-reference - отдельное понятие
  ```
  Однако так писать можно (b и c — ссылки на a):
  ```
  int a = 0;
  int& b = a;
  int& c = b;
  ```
- no arrays of references (any kind of arrays)
  T.е. запрещено:
  ```
  int& a[10];
  std::vector<int&> v;
  ```
- no pointers to references
  T.е. это не скомпилируется:
  ```
  int a = 0;
  int&* b = a;
  ```
  Но так:
  ```
  int a = 0;
  int& b = a;
  int* pb = &b; //не что иное, как указатель на a
  ```
  и так:
  ```
  int* a = new int;
  int*& b = a; //ссылка на указатель - теперь изменение b отражается на a
  ```
  ничто не запрещает.

3) При объявлении ссылки *необходимо* инициализировать объектами, указатели (инициализируются адресами переменных) — не обязательно.
4) Кратко о квалификаторе const.

Более подробно (читать с "Указатели"). Обратить внимание на объявление переменных спиралью[8]
.
Константную ссылку можно создать от любого объекта, но неконстантную ссылку от константного объекта — нельзя.
5) Битая ссылка — ситуация, когда используется ссылка на разрушенный (чаще всего из-за выхода из области видимости) объект. Использование такой ссылки является UB (undefined behaviour).
Пример:

```
int& foo() {
 int a = 4;
 return a;
}

int main() {
 int a = foo(); // может быть чем угодно, хотя, скорее всего, упадёт с
segmentation fault
}
```

Дополнения:

1. Следует помнить, что C++ “за исключением второстепенных деталей содержит язык C как подмножество”, и иногда полезно понимать, какие концепции являются новыми, а какие унаследованы из C. Указатели достались C++ от его предка, а вот ссылок в C не было.
2. Ссылки можно делать полями классов, причем инициализировать их можно как на месте (since C++11):

   ```
   struct C {
    int field = 0;
    int& field_alias = field;
   };
   ```

   Так и в конструкторе:

   ```
   struct C {
    C(int& x) : x(x) {} [9]
    int& x;
   };
   ```

   В одном из этих мест инициализация должна быть обязательно, т.к. ссылка должна быть проинициализирована на момент создания.

2. (done) Расскажите о том, что такое классы, объекты, члены классов, поля, методы, модификаторы доступа, инкапсуляция. Расскажите про конструкторы и деструкторы, операторы присваивания, “правило трех” и “правило пяти”, про генерацию компилятором этих методов. Что такое списки инициализации в конструкторах и делегирующие конструкторы, зачем они нужны? Покажите на примере класса String, как правильно определять вышеперечисленные методы.

1) Класс — это пользовательский тип данных. Переменная класса (в смысле типа) называется объектом класса. Члены класса — данные, связанные с классом. Различают 2 вида членов:

1. Переменные-члены (поля) — переменная, связанная с классом. Все данные объекта хранятся в его полях.
2. Функции-члены (методы) — функции, определенные внутри класса.

---

[8] Если Вам интересно, как читать более сложные объявления, обратитесь к секции 5.12 книги “Язык программирования Си” от Б. Кернигана и Д. Ритчи (K&R) — прим. тов. Титова
[9] Заметим, что если заменить `int& x` на `int x`, то ссылка будет инициализирована временным объектом - ещё один пример битой ссылки.

Модификаторы доступа — ключевые слова, которые определяют набор ограничений на доступ к функциям- и переменным- членам объектов класса. Делят на:

- `public` — доступ открыт всем, кто видит определение данного класса.
- `protected` — доступ открыт классам, производным от данного, и друзьям. То есть производные классы и друзья получают свободный доступ к таким свойствам или метода. Все другие классы такого доступа не имеют.
- `private` — доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса — как функциям, так и классам. Однако производные классы не получают доступа к этим данным совсем. И все другие классы такого доступа не имеют.

Инкапсуляция — механизм упаковки данных и функций в единый компонент (в C++ принято рассматривать инкапсуляцию без сокрытия как неполноценную, поэтому можно добавить, что инкапсуляция позволяет ограничить доступ одних компонентов программы к другим). Один из трёх основных принципов ООП. Позволяет пользователю взаимодействовать с объектом через абстракции, не задумываясь об их реализации.

2.1) Конструкторы и деструкторы — это особые методы класса. Особые потому, что не имеют явно указанного типа возвращаемого значения (даже void-а), не содержат конструкций `return` и генерируются по умолчанию, если программист не определил их явным образом.
Конструктор автоматически вызывается при создании объекта класса. Конструкторы:

★ Всегда должны иметь то же имя, что и класс.

* Пример см. в 7) String

Деструктор автоматически вызывается при разрушении объекта класса. Деструкторы:

★ Всегда должны иметь то же имя, что и класс, со знаком “тильда” (~) в самом начале.
★ Не могут принимать аргументы[10][11][12].

* Пример см. в 7) String

2.2) Оператор — это символ, который сообщает компилятору о необходимости выполнения некоторых математических или логических действий. Операторы присваивания также относятся к специальным функциям-членам, т. к. допускают генерацию по умолчанию. Операторы имеют особый синтаксис объявления — перед названием (символом оператора) добавляется ключевое слово **operator**.

* Пример см. в 7) String

3) Отличия конструктора копирования от оператора присваивания.
Конструктор копирования (перегруженный конструктор, принимающий константную ссылку на класс) используется при инициализации *новых* объектов, тогда как оператор присваивания заменяет содержимое уже *существующих* объектов.

4.1) Правило трёх:
Если класс или структура определяет один из следующих методов, то они должны явным образом определить все три метода:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием

4.2) Правило пяти (расширение правила трёх, появилось с выходом стандарта C++11):
Если класс или структура определяет один из следующих методов, то они должны явным образом определить все пять методов:

---

[10] Как следствие, не допускают перегрузку. Именно поэтому у каждого класса есть только один деструктор.
[11] Даже если очень хочется
[12] Никто не ожидает ~~Испанскую инквизицию~~ третью сноску подряд!

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

5) Генерация специальных функций-членов по умолчанию происходит только при необходимости (т. е. например, если в программе не используется конструктор копирования, то и сгенерирован он не будет) и если соблюдены условия.

6.1) Списки инициализации в конструкторах

При инициализации в области видимости конструктора возникают проблемы с инициализацией полей класса (при входе в область видимости поля уже проинициализированы по умолчанию и при “инициализации” в конструкторе на самом деле происходит копирование полей[13]). Решить это можно с помощью списков инициализации в конструкторах.

Синтаксис:

```
int field;
bool complex;
Integer(int arg, bool complex): field(arg), complex(complex)
{ /*realization*/ }
```

Таким образом можно инициализировать только поля текущего класса.

6.2) Делегирующие конструкторы (since C++11)

В списках инициализации можно вызывать конструкторы (другой конструктор текущего класса или конструктор ближайшего предка). Следует помнить, что:

- В такой конструкции нельзя вызывать несколько конструкторов одного и того же класса.
- В случае вызова конструктора текущего класса все остальные поля класса уже нельзя инициализировать в списке (они могут быть проинициализированы в области конструктора). Если же вызывается конструктор предка, то поля текущего класса таким образом инициализировать можно.

```
int field;
bool complex;
Integer(): field(0) {}
Integer(int arg, bool complex): Integer() { // Integer(), complex(complex)
нельзя
 this->complex = complex;
}
```

Такие конструкторы позволяют писать код, устойчивый к исключениям[14].

7) Пример класса String:
https://gist.github.com/Wutem/b69a10243a0d1f8c639f2cb777e37d4d

Дополнения:

1. Если класс объявлен через ключ. слово `class`, то все члены по умолчанию приватные, если через `struct`, то публичные.
2. Оператор присваивания (=) должен возвращать ссылку на класс (`*this`).

---

[13] Это создаёт проблему, если у класса есть поле-ссылка, которая должна привязываться к объекту в конструкторе. Без списков инициализации в конструкторах провернуть это не удастся.
[14] Устойчивость к исключениям заключается в отсутствии точки, где генерация исключения могла бы привести к утечке памяти (см. билет 10, 4 исключения в конструкторах).

3. Методы можно явно удалять через ключ. слово delete (имеет смысл для особых методов-членов).
```
void foo() = delete;  //не используется, но такая конструкция тоже возможна
String() = delete;
```
4. В C не было ООП (было понятие Plain Old Data (POD) в виде struct, но оно не является полноценным ООП)
5. (advanced) При реализации “правила трёх/пяти” полезно использовать идиому copy-and-swap.

3. (done) Расскажите о том, что такое классы, объекты, члены классов, поля, методы, модификаторы доступа, инкапсуляция. Расскажите о ключевых словах static, explicit, mutable, friend с примерами ситуаций, когда их следует применять.

Ответ на первый пункт см. в предыдущем билете.

1) `static` переменная сохраняет своё значение даже после выхода из блока, в котором она определена. То есть она создаётся (и инициализируется) только один раз, а затем сохраняется на протяжении выполнения *всей* программы.
Static поля часто используются, чтобы не привязываться к объектам класса. (Примеры: Теормин 21)
Static поля классов имеют особенность:

★ Если static поле не помечено const, то его инициализация должна происходить вне тела класса.
★ Если же поле помечено const, то инициализировать его можно как в теле класса, так и вне его.

Пример, иллюстрирующий особенность:
```
class User {
 public:
   User() {
     id = id_counter;
     ++id_counter;
   }
 private:
  int id;
  static int id_counter;
  static const int start_id_number = 0;  //инициализация const static в классе
};

int User::id_counter = User::start_id_number;  //инициализация не const static - вне класса
```

2) `explicit` ставится перед конструктором/оператором приведения типа и запрещает компилятору неявные конверсии с его использованием. Помечать конструкторы классов explicit во избежание непредвиденных преобразований (например, в аргументах функций) — распространённая практика.

```
struct String {
  explicit String(size_t size);
}
```

```
int main() {
  String s1(5);     //ok
  String s2 = 5;    //error
  String s3 = 'a'; //error
}
```

3) mutable снимает константность с поля класса, позволяя изменять его, даже если сам класс помечен const. Часто используется в качестве спецификатора в лямбда-функциях (см. билет 41).

```
struct String {
  String(const char[]);
  mutable size_t debug_number; //это поле можно изменять
}

int main() {
 const String s = "abc";
 s.debug_number += 5;  //ok
}
```

Условия, требуемые квалификатором const, не распространяются на mutable:

```
struct C {
 mutable int a;
 void foo() const {
   a +=1; //ok
 }
};
```

4) Дружественные функции (или классы) — это функции (или классы), которые не являются членами класса, однако имеют доступ ко всему, к чему имеет доступ текущий класс, кроме полей/методов класса, чьим другом является наш класс — дружба не транзитивна. Для определения дружественных функций (или классов) используется ключевое слово friend. Часто используется для перегрузки операторов, когда левый операнд не является членом класса (например, ввод и вывод в поток).

```
class String {
 public:
  String(const char[]);
  //  позволяет определить оператор вывода в поток
  friend std::ostream& operator<<(std::ostream& out, const String& s);
 private:
  char* buffer;
  size_t size;
};

std::ostream& operator<<(std::ostream& out, const String& s) {
 for (int i = 0; i < s.size; ++i) {
   out << s.buffer[i];
 }
 return out;
}

int main() {
 const String s = "abc";
 std::cout << s;
```

```
}
```

Дополнения:

1. Страуструп советует заводить как можно меньше друзей.

4. (done) Расскажите о возможностях перегрузки операторов в C++. Покажите на примере реализации длинной арифметики (класс BigInteger), как правильно перегружать бинарный плюс, унарный и бинарный минус, операторы составного присваивания с плюсом и минусом, операторы сравнения, префиксный и постфиксный инкремент.

1) Оператор — это символ, один из видов токенов в выражении. Синтаксис перегрузки операторов очень похож на определение функции: для этого нужно использовать имя `operator`@, где @ — это лексема оператора (`+`, `-`, `<<`, `>>` и др.). Например:

```
Integer operator+(const Integer& rv) const { /*realization*/ }
```

Существует два способа перегрузки операторов: операторы как глобальные функции, (возможно) дружественные для класса, и как методы самого класса. Любой оператор можно перегрузить любым из этих способов, за исключением:

- `=`
- `()`
- `[]`
- `->`

Они обязательно являются членами класса.

Запрещено:

- Перегружать операторы выбора члена класса (.) и разыменования (`.*`) указателя на член класса (однако `->` и `->*` [15] перегружать можно!), а также тернарный оператор (`?:`) и проход через область видимости (`::`).
- Определять свои операторы (возможны проблемы с определением приоритетов) и изменять приоритеты операторов

Рассмотрим некоторые операторы более подробно.

1.1) `[]`
Во многих STL контейнерах имеется 2 реализации.
Одна для не const контейнеров:

```
int& operator[](int index) { /*realization*/ }
```

Вторая для const:

```
const int& operator[](int index) const { /*realization*/ }
```

1.2) `()`

```
bool operator()(int args) { /*realization*/ }
```

Объекты типа, для которого определён оператор (), называются *функторами*.

1.3) `&& || ,`

[15] Илья упоминал об *указателях на члены* на лекции. В вопросе их нет, и не думаю, что их здесь спросят (это пункт в продвинутых вопросах). Оставлю ссылку для желающих посмотреть на эту штуку.

Стандартом регламентировано, что эти операторы сначала обрабатывают левую часть, и только потом правую. При перегрузке это свойство теряется[16] (поэтому перегружать их не рекомендуется).

1.4) `* & ->`
Перегрузка `*` применяется в умных указателях. `&` — можно перегрузить получение адреса от объекта.
`->` — унарный оператор; возвращаемый объект должен поддерживать операцию `->`.

```
struct A {
 void foo() { std::cout << "Hi" << std::endl; }
};

struct B {
 A a;
 A* operator->() {
   return &a;  // возвращаемый объект должен поддерживать операцию ->
 }
};

int main() {
 B b;
 b->foo(); //напечатает Hi
}
```

1.5) Операторы приведения типов
Операторы приведения типов определяют конвертацию текущего класса в указанный тип. Они поддерживают ключ. слово explicit (см. билет 3), которое запрещает неявную конвертацию:

```
explicit operator int() { // тип возвращаемого значения указывать не нужно
 int a = 10;
 return a;
}
```

Указанный тип может быть и пользовательским:

```
struct A {};
struct B {
 operator A() {/*realization*/}
};
```

➔ Далее за примерами реализаций обращаться к “5) Реализация BigInteger”

2) Операторы составного присваивания (+=, −=, *= и т.д.).
Именно в них рекомендуется реализовать основную логику операции.
Возвращают ссылку на класс.

3) Бинарные операторы (a + b и т.д.).
Бинарные арифм. операторы реализованы через операторы составного присваивания.

★ Их стоит делать *глобальными (дружественными[17]) функциями*. Это позволяет менять местами операнды:

```
5 + BigInteger(3);
```

Если же определить их как методы класса, то левым операндом обязательно должен быть объект класса. Таким образом, конструкция выше в этом случае работать не будет (для int не определён оператор + с правым операндом BigInteger).

[16] Так было до C++17. Начиная с 17-го стандарта, свойство сохраняется и при перегрузке.
[17] Это необязательно. (Make friends with caution.)

4) Префиксный(++a) и постфиксный(a++) инкременты.
Чтобы отличить префиксный инкремент от постфиксного, Страуструп решил добавить в сигнатуру постфиксного инкремента фиктивный параметр int[18].

5) Реализация BigInteger:
https://gist.github.com/Wutem/0d148b23e3933e8e131f43b4fa55baa7

5. (done) Расскажите о наследовании в C++. В чем разница между приватным и публичным наследованием? Каковы правила видимости полей и методов родителя в теле наследника, как явно обратиться из наследника к полям и методам родителя? Каковы правила видимости родителей и их полей и методов при двухуровневом наследовании, как действует слово friend в этих ситуациях? Что такое “срезка при копировании”? Какие неявные конверсии разрешены между родителями и наследниками, в т.ч. ссылками (указателями) на них?

1) Может возникнуть проблема, когда у по факту очень похожих (но всё же разных) классов есть одинаковые методы, которые не хочется писать много раз. Или если хочется сделать массив, например, автомобилей, но для каждой модели есть свой класс, а массив должен содержать одинакового типа объекты. Возникает потребность в каком-то общем для всех классе, в котором будет содержаться общая информация для всех более узких типов. Для этого нужно наследование.
Итак, наследование — это одна из основных концепций ООП, позволяющая создавать классы на основе других классов, при этом заимствуя их функционал. (Как это писать, см. примеры внизу)

2) Но не всегда хочется, чтобы все знали, кто от кого наследован, для этого существуют ключевые слова public, private и protected. Посмотрим, как это работает:

```
class Base {
public:
   int a = 1;
};

class Public_Derived : public Base {
public:
   int Base_a() { return a; }  // Можно, так как наследник
};

class Doble_Derived_Public : public Public_Derived {
   int d_Base_a() { return a; }  // Можно, так как отец public
};

class Protected_Derived : protected Base {
public:
   int Base_a() { return a; }  // Можно, так как наследник
};

class Doble_Derived_Protected : public Protected_Derived {
   int d_Base_a() { return a; }  // Можно, так как отец protected
};

class Private_Derived : private Base {
public:
   int Base_a() { return a; }  // Можно, так как наследник
};
```

[18] Интересно, что int нельзя заменить другим типом.

```
class Doble_Derived_Private : public Private_Derived {
   int d_Base_a() { return a; }  // Нельзя, так как отец private
};


int main() {
   Public_Derived pub;
   Private_Derived prid;
   Protected_Derived prod;

   pub.a = 2;  // Можно, так как public
   prod.a = 2;  // Нельзя, так как protected
   prid.a = 2;  // Нельзя, так как private
   // Так же будет и с любым наследником наследника
   return 0;
}
```

Разберемся, почему так. В целом, суть вот в чём: public наследование позволяет видеть всем, что ты наследник (видеть = иметь доступ к полям/методам), protected позволяется видеть только наследникам (то есть только внутри наследника есть доступ) и друзьям, а private — вообще никому (опять же, кроме друзей).

3) Внутри наследника можно обратиться к public и protected полям/методам родителя (вне, естественно, только к public). А что будет, если есть одинаково названная функция в родителе и наследнике? Тогда правила таковы: из объекта класса наследника будет всегда пытаться вызваться именно функция из наследника, а если не получается, то ошибка компиляции. То есть переопределение функции в наследнике как бы замещает родительскую полностью. Пример:

```
class Base {
public:
   void f() {}
   void g(int a) {}
};

class Derived : public Base {
public:
   void f() {}
   void g() {}
};

int main() {
   Derived d;
   d.f(); // Вызовется та, что из Derived
   d.g(3); // Ошибка компиляции, так как он уже выбрал другую g, а ей не
нужны аргументы

   return 0;
}
```

А что, если мы всё же хотим именно ту функцию, что в родителе? Тогда делаем так:

```
   Derived d;
   d.Base::f(); // Всё хорошо
   d.Base::g(3); // И тут тоже
```

```
    // И понятно, что если наследование не public, то так нельзя
```

(Также можно написать в наследнике Using Base::g, и тогда та g будет рассматриваться при “разрешении перегрузок”)
Полезно отметить, что даже если в предпредыдущем примере функция g в наследнике была бы приватной, то всё равно была бы ошибка компиляции, потому что проверка доступа происходит после разрешения имён.

4) Разберёмся получше с двухуровневым наследованием.

```
struct Granny {};

struct Mom: private Granny {};

struct Son: public Mom {
    void f() {
        Mom m;
        Granny g;  // Ошибка компиляции
    }
    void g(Granny g) {} // И даже здесь
};
```

Так происходит, потому что в контексте Son вам вся бабушка сама запрещена, но можно обратиться к Granny из глобальной области видимости (то есть ::Granny)
Существует ключевое слово friend, которое позволяет объявлять “друзей” класса, то есть функции/классы/etc, которые имеют доступ ко всему(!), к чему имеет доступ наш класс (разве что кроме полей/методов класса, чьим другом является наш класс. Дружба не транзитивна). Вот как это делается

```
struct Granny {
    friend class Son; // Объявление Son другом
    friend void q();  // Объявление функции q другом
};
```

Но, к сожалению, (или к счастью) даже это не поможет нам из Сына обратиться к Бабушке, потому что нам запрещены не поля, а сама Бабушка. Но если мы сделаем Сына другом Мамы, то всё будет хорошо, потому что друзьям мамы можно всё, что и Маме. До этого именно Мама запрещала обращаться к Бабушке, а теперь она стала вашим другом, поэтому запрет снят.

5) Рассмотрим случай:

```
struct Base {};

struct Derived: Base {};

int main() {
    Base b;
    Derived d;
    b = d; // Так можно
    d = b; // Так нельзя
    return 0;
}
```

Разберёмся. Действительно, так как Base — это частный случай Derived, а не наоборот, всё получается так, как получается. Но возникает так называемая срезка при копировании, то есть в b будет лежать как бы обрезанная часть d, та часть, которая пришла ему от класса родителя.

6) Также разрешены такие конверсии: (Подробнее в билете номер 8)

```
struct Base { int a; };
struct Derived: Base { int b; };

int main() {
   Derived d;
   Base& b = d;  // При этом через b доступ есть только к Base-части d, но мы-то знаем, что на самом деле там d
   Base* bb = &d; // Аналогично. И поэтому мы можем использовать cast, чтобы вновь иметь дело с наследником
   Derived dd = static_cast<Derived&>(b);
   Derived* ddd = static_cast<Derived*>(bb);

   Base q;
   Derived w;
   static_cast<Base>(w).a;  // Можно. Тоже срезка при копировании
   static_cast<Derived&>(q).b;  // Скомпилируется, будет плохо, скорее всего обратится к чужой памяти (UB)
   static_cast<Derived>(q).b;  // А так нельзя вообще никак, не определив самим, видимо, это преобразование
// И, опять же, всё это работает только при публичном наследовании
// (Но при private можно reinterpret_cast'ом [кроме последней строчки])
   return 0;
}
```

Дополнения:

1. Если не писать модификатор доступа (private, public, etc), то по умолчанию наследование будет публичным для структуры (struct) и приватным для класса (class).
2. Рубрика “Забавные примеры” (спасибо товарищу Илье Баюку):

```
struct Base {
   int a = 4;
};

struct Private_Derived : private Base {
   friend struct C;
};

class Doble_Derived_Private : public Private_Derived {};

struct C {
   Doble_Derived_Private q;
   void print() { std::cout << q.a; }  // И это работает!
};

int main() {
   C c;
   c.print();
   return 0;
}
```

Этот пример показывает, насколько сильна дружба! В целом, это выглядит так: вам мама не говорит, кто ваш дедушка, но все её друзья это знают.[19]

6. (review) Как размещаются в памяти объекты классов-наследников? В каком порядке вызываются конструкторы и деструкторы при наследовании, как инициализировать поля родителя при определении конструктора наследника? Что такое множественное наследование, в чем заключается проблема ромбовидного наследования, что такое виртуальное наследование? Что означает и как возникает ошибка “ambiguous base”? Что означает и как возникает “warning: inaccessible base class”? Что происходит при комбинации виртуального и невиртуального наследования?

1) Если я правильно понял вопрос, то в памяти просто подряд идут все поля, начиная с родительских и заканчивая нашим. (Если унаследован от нескольких, то в том порядке, в котором унаследован скорее всего)

2) Конструкторы вызываются в порядке от самого дальнего предка до нас, деструкторы — наоборот. Важно помнить, что если у родителя нет конструктора по умолчанию, то мы должны явно инициализировать “родительскую” часть нашего класса:

```
struct A {
   int a;
   A(int a) : a(a) {}
};

struct B : A {
   int b;
   //B(int b) : b(b) {}  // Нельзя, потому что у A нет конструктора по
умолчанию
   //B(int a, int b) : a(a), b(b) {} // И так тоже нельзя, потому что поля
инициализировать можно только свои
   B(int b) : A(3), b(b) {}
};
```

К слову, так можно инициализировать ближайших предков, более дальних нельзя (при невиртуальном наследовании)

3) Множественное наследование — наследование от нескольких классов. В связи с этим возникает проблема, которая называется “проблема ромбовидного наследования”:

```
struct A {
   int a;
   int f() {}
};

struct B1 : A {};
struct B2 : A {};
struct C : B1, B2 {};

int main() {
   C c; // Создали объект класса C
   c.a; // (1) Нельзя (ошибка компиляции)
   c.f(); // (2) Нельзя
   c.B1::f(); // (3) Можно
```

[19] Люблю плюсы

```
    return 0;
}
```

Что же не так? А проблема заключается в том, что C унаследован от B1 и B2, унаследованных от A (в памяти лежит приблизительно так: A_B1_A_B2_C), и получается как бы два A. То есть в выражениях (1) и (2) неизвестно, к каким именно a и f мы обращаемся, которые от того A, что от B1 или от того A, что от B2. А в выражении (3) всё хорошо, потому что однозначно. Поэтому в некоторых компаниях множественные наследования запрещены. (Это ошибка “Ambiguous base class”)
(Заметим, что не имеет значения, как именно был унаследован: public, private или protected)
Здесь же можно сказать о том, что если у вас B унаследован от A, и C унаследован от A и B, то вы опять же не сможете обращаться к полям A через объект класса C. В таких случаях возникает “warning: inaccessible base class”
И на помощь к нам приходит виртуальное наследование. Оно как раз и делает то, что нужно, то есть не создает “лишних бабушек”.

```
struct A {
    A(int i) { std::cout << "A" << i << "\n"; }
};

struct B1 : virtual public A {
    B1() : A(1) { std::cout << "B1\n"; }
};
struct B2 : public virtual A {  // Видимо, порядок не важен
    B2() : A(2) { std::cout << "B2\n"; }
};
struct C : B1, B2 {
    C() : A(0) { std::cout << "C\n"; }  // А теперь обязательно здесь нужно
инициализировать A
};
```

(при создании выведется “A0 B1 B2 C”)
Отметим последнюю строчку. Работает это так: если есть виртуальное наследование, то при создании C он сначала создает собственный A для всех виртуально унаследовавшихся, а уже потом, как и раньше, начинает создавать прямых родителей (при этом те из них, которые унаследовались НЕ виртуально, создадут каждый свой A).
Для проверки себя, подумайте, что бы вывелось, если B1 или B2 унаследовались бы не виртуально. (ответ ниже)

Дополнения:

1. В случае виртуального наследования в классе хранится еще указатель на родителя (потому что он должен быть общий для всех)
2. Если при ромбовидном наследовании пометить всё как виртуальное и переопределить поле бабушки в одном(!) из родителей, то ошибки не будет, и вызовется переопределённое поле (функция)

```
struct A {
    int a;
};

struct B1 : virtual A {
    int a;
};
struct B2 : virtual A {
```

```
    // int a;  // А вот если тут еще написать, то опять будет
неоднозначность
};
struct C : B1, B2 {};

int main() {
    C c;
    c.a;  // Не ошибка
    return 0;
}
```

3. Ответы:
    a. Если B1 не виртуально, то “A0 A1 B1 B2 C”
    b. Если B2 не виртуально, то “A0 B1 A2 B2 C”
4. (advanced) О размещении структур (в частности, их полей) в памяти есть несколько интересных моментов.
5. Размер класса — сумма размеров полей всех предков, своих полей и всех виртуальных таблиц (то бишь + размер n указателей, где n — количество полиморфных классов в иерархии наследования (разумеется, от тебя и выше))
6. Размер пустого класса — хотя бы 1 байт (чтобы у разных объектов одинакового типа были разные адреса)
7. Кому интересно, есть такая штука, как empty base optimization

7. (done) Что такое виртуальные функции? В чем разница между виртуальными и невиртуальными функциями при наследовании? Что такое таблица виртуальных функций? Что такое абстрактные классы и чисто виртуальные функции? Для чего нужен виртуальный деструктор? Что такое полиморфизм и как это понятие в C++ связано с виртуальными функциями?

1) Допустим, у нас есть класс-родитель (“Предмет”) и наследники (“Матан”, “Линал” и т.д.). Мы хотим определить функцию-член “Скатать Д/З”. В “Предмете” определяем общий алгоритм для всех предметов, а в каждом наследнике переопределяем эту функцию (Например, мы знаем, что некто Святослав[20] шарит в физосе, и быстрее сразу к нему обратиться, чем выполнять общий алгоритм). И также у нас есть просто функция, которая принимает ссылку на “Предмет” и делает что-то, в том числе вызывает “Скатать Д/З”.

```
struct A {
    void f() {}
};

struct B : A {
    void f() {}
};

void g(A& a) {
    a.f(); // (1)
    // ...
}
```

И это не то, что нам нужно, потому что в (1) будет всегда вызываться та f, что из A. На самом деле мы же хотим, чтобы для каждого наследника в данном случае вызывалась своя функция, более специализированная. Для этого существуют виртуальные функции и ключевое слово virtual. Достаточно написать `virtual void f() {}` в классе-родителе, и, вуаля, всё работает, как мы и хотели. В принципе, в этом и заключается отличие виртуальных функций от не виртуальных

[20] Man I really wonder who *that* is

2) Таблица виртуальных функций — это условно “таблица”, в которой хранится информация, какая версия функции предназначена для какого объекта. То есть для каждой из виртуальных функций там находится что-то типа “тип наследника — адрес нужной реализации функции”. И указатель на эту таблицу хранится в полиморфном классе аналогично полям.

3) Теперь рассмотрим похожую ситуацию, как в п.1. Пусть у нас есть геометрические фигуры, и мы хотим сделать класс-родитель “Фигура”, чтобы создать, допустим, массив “указателей на Фигуру” и пихать туда разные фигуры. И, допустим, мы хотим уметь искать площадь фигур. Но нет общего алгоритма подсчёта площади любой фигуры, который мы могли бы написать в “Фигуре”. На помощь приходят чисто виртуальные функции. Мы пишем `virtual double area() = 0;` в родителе, и такая функция *может не иметь*[21] тела, и её обязаны определить наследники. Тогда класс-родитель становится абстрактным классом (то есть, нельзя создать объект такого типа), и всякий наследник, который не определит эту функцию, будет тоже абстрактным классом (также в некоторых языках такие классы называются *интерфейсами*[22]).

4) Сразу решим ещё одну проблему. Представим следующую ситуацию:

```
// B - наследник A, деструктор обычный
B* b = new B();
A* a = b;
delete a;
```

Произойдет утечка памяти, потому что вызовется деструктор A, и B-часть останется неудалённой. Поэтому при наследовании всегда стоит объявлять деструктор виртуальным, если вы хотите обращаться к наследнику через указатель на родителя

5) Полиморфизм — ещё одна из основных концепций ООП, возможность иметь много реализаций одного интерфейса. И при обращении к методу будет вызываться нужная реализация (в зависимости от того, что передали в функцию). В данном случае это вырождается как раз в то, что у нас есть функция в полиморфном классе и точно такая же в наследниках (имеется в виду, с точно такой же сигнатурой), и при вызове оной у объекта будет вызываться та функция, которая нам нужна.
(Классы, у которых есть виртуальные функции, называются *полиморфными*.)

Дополнения:

1. Если, например, в абстрактном классе функция объявлена const, а в наследнике не const, то это будет не переопределение, а новая функция. Чтобы застраховаться, есть ключевое слово override (since C++11), которое на этапе компиляции проверит корректность переопределения:
   `void f() const override {};`
2. Если вы не хотите, чтобы от вас наследовались, или чтобы наследники переопределяли виртуальную функцию, то можете использовать ключевое слово final (since C++11):
   `struct A final {}`

   `virtual void f() final {} //такую функцию нельзя переопределять в наследниках`
3. Также стоит понимать, что выбор нужной функции происходит в runtime, а не на этапе компиляции.
4. Из-за того, что полиморфные классы должны обращаться к таблице виртуальных функций, они работают относительно медленно (и требуют немалый объём памяти).

---

[21] Именно так: не “не может иметь”, а “может не иметь”!
[22] В C#, например, есть как абстрактные классы, так и интерфейсы.

Любопытный факт: слова override и final разрешены к использованию в качестве идентификаторов.[23]

8. (done) Расскажите о разновидностях приведений типов в C++. В чем разница между C-style cast, static_cast, const_cast, dynamic_cast и reinterpret_cast, когда они применимы, а когда нет? Для чего нужен каждый из этих кастов?

Кроме стандартных приведений типов (тех, что выполняют преобразования между встроенными типами), можно определить:

1) конструктор для пользовательского типа:

```
struct Integer {
 int value;
 Integer(int value): value(value) {}
 explicit Integer(const std::string& s): value(s.size()) {}
};
```

Конструкторы, помеченные как explicit, не могут участвовать в неявных преобразованиях.

2) Оператор преобразования типа:

```
struct Integer {
 int value;

 operator int() const {
   return value;
 }

 // Можно определять для пользовательских типов:
 explicit operator std::string() const {
   return "";
 }
};
```

Начиная с C++11 к операторам преобразования типа применимы те же правила explicit/implicit, что и к конструкторам.

3) Пользовательские литералы, в каком-то смысле, тоже про это: https://habr.com/ru/post/140357/

Для явных преобразований есть cast-операторы:

1) static_cast: принимает решение на этапе компиляции и может, в частности:
   a) Вызвать конструктор или определённый пользователем оператор преобразования типа — в частности, помеченный как explicit
   b) Преобразовать тип указателя или ссылки в случае наследования и ряде других
   c) Использовать стандартное преобразование типа.
   d) Как частный случай пункта (a), при наследовании возможно одностороннее преобразование Derived → Base. Derived неявно преобразуется в const Derived &; далее, благодаря полиморфизму, в const Base &; после чего будет вызван конструктор копирования Base. При этом произойдёт срезка: потеряются поля наследника. Обратное преобразование невозможно, если явно не написан конструктор Derived(const Base &). Иллюстрация:

```
Base b;
Derived d;

static_cast<Base>(d);
```

[23] Шок

```
//ok: Вызывается конструктор копирования Base, Derived& неявно
преобразуется к const Base&. (Конструктор сделает срезку)

static_cast<Derived>(b);
//error: Здесь хотел бы быть вызов конструктора копирования
Derived, но ссылка на родителя не преобразуется в ссылку на
потомка неявно (Base& !-> Derived&) => подходящей перегрузки
конструктора нет. Вероятно, это бы работало, если бы мы
определили свой конструктор Derived(const Base&)

static_cast<Base&>(d);
//ok: Ссылка на потомка преобразуется в ссылку на родителя даже
неявно

static_cast<Derived&>(b);
//ok: ссылку на родителя можно преобразовать в ссылку на
потомка, но явно
```

Проверка фактического типа аргумента не выполняется

```
Base* x;
Derived* y = static_cast<Derived*>(x); // OK
```

2) reinterpret_cast: *более топорно* меняет тип выражения. Не выполняет никаких дополнительных операций в рантайме.
Разрешаются любые преобразования указателей, не понижающие константность.
Благодаря этому, в отличие от static_cast, можно преобразовывать указатель на наследника к родителю при private наследовании:

```
struct Base {
 int x = 18;
};

struct Derived : private Base {
 int y = 97;
};
Derived* d = new Derived;
Base* b = reinterpret_cast<Base*>(d); // OK, хотя со static_cast было
бы CE
```

reinterpret_cast просто говорит что-то вроде “теперь под этим участком памяти теперь лежит вещь такого типа, давайте обращаться с ней по-другому”. Использовать на свой страх и риск[24].
Применяется при приведении указателей одного типа к другому.[25]

(Замечание от тов. Титова:
В C++20 завезут bit_cast, который настоятельно рекомендуется использовать для тривиально копируемых типов одинакового размера.)

3) const_cast: единственный ~~волшебник~~ каст, который может кастовать сквозь const

```
int x = 5;
const int& y = x;
int& z = const_cast<int&>(y);
```

[24] Никогда
[25] Сноска сверху немного врёт

const_cast'ом мы говорим компилятору "слушай, этот объект на самом деле не константный, давай ты не будешь его считать таковым". Если объект действительно изначально был не константным, то всё будет хорошо, как в коде выше. Но если объект изначально был константой…

```
const int x = 5;
int& z = const_cast<int&>(x);
```

...то получится UB, так как константы хранятся в какой-то особой области памяти с особыми правами доступа

4) dynamic_cast: выполняет преобразования между наследниками и потомками, во время выполнения проверяя фактический тип аргумента.
Если тип не соответствует, бросает исключение std::bad_cast.
Данный код не компилируется, так как чтобы кастовать Base → Derived, Base должен быть полиморфным

```
struct Shape {};
struct Ball : Shape {};
void foo(Shape& shape) {
    Ball& ball = dynamic_cast<Ball&>(shape);
}
```

Это скомпилируется:

```
struct Shape {
     virtual void foo(){}
};
struct Ball : Shape {};
void foo(Shape& shape) {
    Ball& ball = dynamic_cast<Ball&>(shape);
    // Если фактический тип shape не является Ball или его
наследником, выбросит исключение}
```

При этом каст вверх — то есть от Derived к Base — не нуждается в полиморфизме

5) C-style cast: комбинация перечисленных выше преобразований. Самый жёсткий из всех кастов, так как он не явно перебирает все комбинации кастов, в итоге деградируя до reinterpret_cast. Вообще никогда не надо использовать.

9. (done) Расскажите о шаблонах в C++. Что такое инстанцирование шаблонов, специализация шаблонов? Какие 3 вида шаблонов существуют? Какие 3 вида шаблонных параметров существуют? Как использовать шаблоны с переменным количеством аргументов и оператор sizeof…? Что такое Curiously Recurring Template Pattern? Что такое полиморфизм, и как это понятие в C++ связано с шаблонами?

1) Инстанцирование шаблона — процесс создания конкретного класса/функции/using (см. далее) из шаблона путем подстановки аргументов. Процесс инстанцирования шаблонов происходит между препроцессингом и компиляцией. Происходит с проверкой типов аргументов (совместимость по присваиванию, по приведению типов). Для классов нужно явное указание параметров[26].
2) Специализации шаблонов нужны для случаев, когда мы хотим, чтобы с данным набором типов данных функция вела себя по-другому (например, со строками выводила дополнительно "string"). Пример кода:

```
template <typename T>
void Print(const T &x) { std::cout << x << '\n'; } // general

template <>
```

[26] Как писалось в теорминимуме, начиная с C++17, это не совсем так, и для классов в некотором случае тип может выводиться автоматически

```
void Print<std::string> (const std::string &x) { std::cout << "string: " << x << '\n'; } // for strings only

int main() {
   Print(10); //prints 10
   Print<std::string>("Hello"); // prints “string: Hello”
}
```

Здесь, однако, есть нюанс: существует понятие “неполная (частичная, partial) шаблонная подстановка/специализация” — это специализация некоторых (но не всех!) шаблонных параметров. В классах это работает корректно:

```
template<class U, class V>
struct C {};

// specialization
template<class U>
struct C<U, int> {};
```

Но в функциях это вызывает CE: function template partial specialization is not allowed:

```
template<class U, class V>
int f();

// specialization
template<class U>
int f<U, int> (); // error
```

Причем следующий код работает, т.к. это уже не специализация, а перегрузка:

```
template<class U, class V>
void f() {}

// overload
template<class U, int>
void f() {}
```

Дополнение: при написании частичной специализации шаблона класса нельзя указывать значения параметров по умолчанию (они будут взяты из общей версии).

3) Виды шаблонов: шаблонные классы, функции, using. Пример кода для using:

```
template<typename T, typename V>
struct CVeryLongTypeName {
   void f() {
       std::cout << "MyClass";
   }
};

template<typename T, typename V>
using CMy = CVeryLongTypeName<T, V>;

int main() {
   CMy<int,double> my;
   my.f(); // prints MyClass
}
```

4) Виды шаблонных параметров: template template parameters, non-type template parameters, type (в примере идут в том же порядке); Пример кода:

```
template <class Type>
struct SA {
```

```
 Type data;

 SA(int x) : data(x) {}
 SA(double x) : data(x) {}
};

//template template, non-type template, type (with default type)
template<template<typename> class Container, int n, typename T = int>
void f() {
 Container<T> arr(n);
 std::cout << arr.data << ' ' << typeid(T).name() << '\n';
}

int main() {
 f<SA, 10>(); // prints 10 i
 f<SA, 6, double>(); // prints 6 d
}
```

5) sizeof... — это специальный оператор, введённый в язык только для того, чтобы узнавать число аргументов в паке аргументов (parameter pack), когда их переменное количество. Пример кода с variadic templates:

```
// Base
void f() {
 std::cout << "end";
}

// Recursive
template<typename Head, typename... Tail>
void f(const Head& h, const Tail&... tail) {
 std::cout << h << ' ' << sizeof...(Tail) << '\n';
 f(tail...);
}

int main() {
 int x = 0;
 char y = 'a';
 std::string s = "str";
 f(x, y, s); /* prints 0 2
                a 1
                str 0
                end */
}
```

6) CRTP (curiously recurring template pattern) эффективно эмулирует систему виртуальных функций во время компиляции (другими словами, позволяет обращаться из базового класса к производному), но не позволяет делать этого во время выполнения. Пример кода:

```
template<typename T>
struct Base {
   void f() {
       std::cout << "B" << ' ';
       static_cast<T*>(this)->f();
   }
};

struct Derived : public Base<Derived> {
   void f() { std::cout << "D" << ' '; }
};
```

```
int main() {
   Derived d;
   Base<Derived> *b = &d;
   b->f(); // prints B D
}
```
Заметим, что никакой рекурсии нет, так как на этапе компиляции происходит только подстановка шаблонов, а сама генерация типов происходит по мере надобности.

7) Полиморфизм — способность функции обрабатывать данные разных типов (т.е. это возможность писать универсальный код для объектов разных типов). Бьёрн Страуструп определил полиморфизм как «*один интерфейс — много реализаций*». Действительно, шаблоны — это статическое проявление полиморфизма, позволяющие объявлять (обобщённо) вещи универсальные для всех типов.
```
template <typename T>
void swap(T& a, T& b){
   T t = std::move(a);
   a = std::move(b);
   a = std::move(t);
}
```

Дополнения:

1. Не забываем про `typename C<T>::type x;` (по умолчанию без инстанцирования компилятор считает всё полями)
2. typedef [что] [чем], using [чем] = [что]
3. Общее правило вывода типов: void f(T&) — отбрасываются неуниверсальные ссылки (const сохраняется), void f(T) — ссылки и const отбрасываются. Пример кода для наглядности:
```
template <typename T>
struct SC {
 SC() = delete;
};

template <typename T>
void f(T& x) {
 SC<T> s;
}

int main() {
 int x = 1; f(x); // T == int
 int& y = x; f(y); // T == int
 const int z = 2; f(z); // T == const int
 const int& t = z; f(t); // T == const int
}
```

10. (done) Расскажите об исключениях в C++. В чем их идея? Как пользоваться оператором throw и конструкцией try...catch? В чем разница между исключениями и ошибками времени выполнения? Что такое спецификации исключений, как пользоваться оператором и спецификатором noexcept? В чем особенности и проблемы исключений в конструкторах и деструкторах?

1.1) Исключение — нестандартная ситуация, возникающая в ходе выполнения программы.
Механизм генерации исключений в C++ реализован с помощью оператора
throw, обработки — с помощью ключ. слов try и catch.
1.2) throw
Тип выражения, указанного в операторе throw, определяет тип исключительной ситуации, а значение выражения может быть передано обработчику (catch).

Стандартные операторы, генерирующие исключения: new и dynamic_cast

1.3) try-catch

Блок кода, который потенциально выкидывает исключение, можно “обернуть” в try-блок, который при возникновении исключения сообщит об этом catch-блоку, следующему сразу за try-блоком:

```
try {
 // код, бросающий исключение
} catch(/*тип исключения*/) {
 // обработка исключения
}
```

Последний, в свою очередь, проверяет исключение на соответствие (по типу), и, если нашёлся подходящий обработчик, исключение считается обработанным. Если же ни один из обработчиков не смог обработать исключение, оно пробрасывается дальше на более высокий уровень (т.е. программа как бы начинает “выходить” из вложенных блоков). После обработки исключения программа продолжит работу с кода, следующего после последнего catch-блока. Если исключение не было обработано на последнем (глобальном) уровне, то программа завершается с RE.

Пример кода с исключением:

```
void ReallyBadFoo() {
 throw 1;
}

void BadFoo() {
 try {
   ReallyBadFoo();
 } catch(int bad_int) {
   std::cout << "BadFoo's handled int exception with value of: " << bad_int
<< '\n';
 }
 std::cout << "BadFoo code after catch-block\n";
}

int main() {
 try {
   BadFoo();
 } catch(char) {
   std::cout << "main's handled char exception\n";
 } catch(double) {
   std::cout << "main's handled double exception\n";
 }
 std::cout << "main code after catch-block\n";
}
```

1.4) Подробнее о catch.

catch ловит исключение по его типу:

```
catch(/*тип выражения исключения*/)
```

Получить доступ к самому выражению позволяет конструкция:

```
catch(/*тип*/ /*имя*/)
```

Например:

```
catch(std::string error) { // если брошена строка, то её значение будет
записано в error
 std::cout << error;
}
```

Также можно поймать любое исключение используя … :

```
catch(...) { /*do something*/ }
```

Исключения можно пробрасывать дальше с помощью “пустого” `throw`[27], причем это исключение может быть поймано только catch-блоками более высоких уровней:

```
catch(...) {
 // do something
 throw;  // исключение, пойманное данным catch, полетит дальше
}
```

Не запрещено, однако, генерировать и новые исключения в catch-блоках.

Важно:

- ★ При обработке исключений компилятор *не делает приведение типов*, *кроме приведения типов между родителем и наследником (и по указателю, и по ссылке) (ну, и указатели можно кастовать к void*)*
- ★ Из всех catch выбирается *единственный* подходящий по порядку, остальные игнорируются.

2) Разница между исключениями и ошибками времени выполнения.
“Не всякое исключение есть RE, и не всякое RE есть исключение”.
Не всякое исключение есть RE.
Ясно, что механизм исключений может использоваться не только для нештатных случаев. Некоторые программисты используют его, чтобы выйти из глубоких уровней вложенности циклов: например, есть 4 цикла, и надо при некотором условии попасть из самого глубокого (4-го) во 2-й. Для этого можно кинуть исключение в 4-м и поймать его во 2-м.[28]
Не всякое RE есть исключение.
RE — критическая ситуация, которая не предусматривает обработки, а сразу приводит к аварийному завершению программы. Обработать можно только то, что было брошено throw! Пример RE: segmentation fault (например, обращение к 100-му элементу массива из 10 элементов — эту ошибку нельзя поймать). А вот операторы new и dynamic_cast по своей задумке могут столкнуться с проблемами, для которых можно предусмотреть решение. Поэтому они кидают исключения (std::bad_alloc и std::bad_cast соответственно). Например, как вариант обработки исключительной ситуации, new может дать возможность стереть наименее важную область памяти под нужды нового new, а dynamic_cast — просто отказаться от cast, если проверка закончилась неуспешно.

3) Спецификации исключений
С помощью спецификации исключений в объявлении функции указывается множество исключений, которые она может возбуждать прямо или косвенно. Спецификация позволяет гарантировать, что функция *не* бросит *не* перечисленные в ней исключения.
До C++ 11 использовался синтаксис:

```
void foo() throw(/*типы*/) { /*do something*/ } // бросает только перечисленные типы
```

Где в *ключевом слове* throw() перечисляются все типы, которые данная функция может выбросить. Однако с C++ 11 конструкция считается deprecated (кстати, ошибка о неверно брошеном исключении — RE). И с 2011 года используется ключ. слово noexcept, которое говорит, что данная функция не бросает исключений вообще (а если бросает, то это (!!!) также RE):

```
void foo() noexcept { /*do something*/ }
```

- ★ *Все функции, не бросающие исключения, стоит помечать noexcept*
  (хотя бы потому, что это ускоряет время работы). Например, стандартные алгоритмы и контейнеры не должны использовать move assignment и move construction, если эти методы

---

[27] Пустой throw также возможно использовать *вне* catch-блоков. В этом случае программа сразу вылетает с terminate() (ибо такой throw нельзя поймать)
[28] Но не надо так делать

могут кидать исключения, т.к. иначе такое поведение противоречит строгой гарантии безопасности исключений.

Возможна ситуация, когда функция бросает исключение в зависимости от каких-то обстоятельств. В этом случае функция всё ещё имеет шанс на noexcept. Тут в игру вступает *оператор* noexcept — bool оператор, который возвращает false, если выражение под ним потенциально бросает исключение, и true в противном случае. Таким образом, если “исключительность” функции f зависит от того, бросает ли исключение функция g, пишется:

```
void f() noexcept(noexcept(g())) { /*do something*/ } //первый noexcept - спецификатор, второй - оператор
```

Причём результат должен быть посчитан на этапе компиляции.

4) Исключения в конструкторах и деструкторах

Если в конструкторе класса вылетело исключение, то его деструктор не будет вызван (логично, т.к. объект же не создался до конца, а частичных деструкторов нет). Это может привести к утечке ресурсов. И вообще, это лишь частный случай более общей проблемы утечки ресурсов: если захват и освобождение ресурсов разделены кодом, потенциально бросающим исключение, то возможна утечка ресурсов. Один из вариантов решения — умные указатели[29]. (Также стоит вспомнить про делегирующие конструкторы.)

Если деструктор класса потенциально бросает исключение, то возможна ситуация, когда по мере разрушения локальных объектов (из-за брошенного исключения), очередной разрушаемый объект кидает ещё одно исключение (из деструктора). Т.е. во время обработки исключения было выброшено ещё одно исключение. Такое поведение сразу обрывается terminate(), т.е. программа вылетает. Начиная с C++11, деструктор неявно помечен noexcept, поэтому, чтобы кинуть исключение (без гарантированного std::terminate), надо явно указать `noexcept(false)`:

```
~C() noexcept(false) {
 throw 1;
}
```

Дополнения:

1. О копировании в исключениях: оператор throw создаёт копию (вызывает конструктор копирования[30]) бросаемого объекта. Единственный случай, при котором throw не делает копии — это пустой throw в catch-блоке (этот же объект полетит дальше). Catch в плане копий работает подобно функциям (может принимать по ссылке или по значению).
2. О функции std::terminate. Она вызывается при ошибках, связанных с исключениями (двойной throw, необработанное исключение и т.д.). Внутри себя она вызывает C-функцию std::abort, которая завершает программу. terminate можно переопределить с помощью std::set_terminate(), например:

```
int main() {
 std::set_terminate([](){ std::cout << "Unhandled exception\n";
std::abort(); }); // лямбда выражения см. в 41 билете
 throw 1;
}
```

11. (review) Расскажите о контейнере std::vector. Предложите реализацию конструкторов, деструктора, операторов присваивания, оператора [], методов at, resize, reserve, front, back для этого класса (с правильной поддержкой аллокаторов и move-семантики).

1) vector — динамический массив, то есть массив непостоянного размера, в котором можно “класть” новый элемент сверху, обращаться по индексу, изменять по индексу и т.д.

---

[29] Это одна из основных причин создания умных указателей

[30] Как следствие, пробрасывается всегда lvalue, причём принимать исключение по && не только не имеет смысла, но даже запрещено.

Пример реализации методов, упомянутых выше:

```
template <class T, class Alloc = std::allocator<T>>
class vector {
public:
    vector( const Alloc& _alloc = Alloc() ): sz(0), capacity(8),
data(traits::allocate(alloc, 8)), alloc(_alloc) {};
    vector(size_t _size, const T& value = T(), const Alloc& _alloc =
Alloc());
    vector(const vector& other);
    vector(vector&& other);
    // Разумеется, есть ещё куча перегрузок

    ~vector() { destruct(); }

    vector& operator=(const vector& other) &;
    vector& operator=(vector&& other) &;

   void push_back(const T& value);
   void pop_back();
   void resize(size_t count);  // Либо дополняет, либо срезает до нужного
size'a.
   void reserve(size_t count); // Увеличивает объём аллоцированной памяти

   size_t size() const { return sz; }

   T& operator[](int index) { return data[index]; };
   T& at(int index);
   T& front() { return data[0]; };
   T& back() { return data[sz-1]; };
   // Разумеется, для этих четырёх нужны ещё const-аналоги, т.е const T&
front() const {...}; и тд

private:
   using traits = std::allocator_traits<Alloc>;

   size_t sz;
   size_t capacity;
   Alloc alloc;
   T* data;

   void destruct();  // Вызывает destruct каждого элемента и деаллоцирует
память
};

template <class T, class Alloc>
vector<T, Alloc>::vector(size_t _size, const T& value, const Alloc& _alloc)
       : sz(_size),
         capacity(_size),
         alloc(_alloc),
         data(traits::allocate(alloc, _size)){
   for (int i = 0; i < _size; ++i) {
       traits::construct(alloc, data + i, value);
   }
}
```

Это была демо-версия, полный код ~~купить~~ смотреть на
https://gist.github.com/Potyashin/d39674b989f8c266510fbbe0f2c1226d

Разберёмся, ~~что за фигня тут написана~~ как работают непонятные функции:

(1) select_on_container_copy_construction(const Alloc& a). Её можно переопределить в своем аллокаторе. Она используется в конструкторе копирования контейнера (см. пример выше) и должна возвращать тот аллокатор, который будет использоваться в объекте, который конструируется копированием. То есть, она *как бы* инкапсулирует конструктор копирования аллокатора. Если мы её не переопределили, то она просто возвращает тот же самый аллокатор `a`. Зачем это надо? Ну, а вдруг у нас какой-то свой хитрый аллокатор. :)

(2) propagate_on_container_copy_assignment. Эта штука говорит нам, хотим ли мы менять аллокатор при присваивании копированием, или нет. То есть, если -//-::value == true, то мы копируем, иначе используем старый. По умолчанию она false.

(3) propagate_on_container_move_assignment. Почти то же самое, что и предыдущая. Если она false (значит, мы не хотим, чтобы аллокатор текущего вектора менялся), и аллокаторы наш и чужой различны, то мы уничтожаем содержимое текущего объекта и move-присваиваем нашему чужой (при надобности расширяя память). Если она false, и аллокаторы равны, тогда мы уничтожаем старое содержимое, деаллоцируем старую память и просто берём поля чужого объекта себе. А если она true, то мы делаем то же самое, только в нужном месте просто move-присваиваем тот аллокатор.

Дополнения:

1. std::allocator_traits находится в <memory>
2. Метод at() бросает исключение std::out_of_range, если вы вылезли за размеры
3. Расставить const'ы и noexcept'ы оставим читателю в качестве упражнения. :)
4. Если есть какие-то вопросы/исправления (особенно если что-то концептуально важное), пишите мне
5. Версия вектора с итератором (см билет 20) (отдельно, ну мб так удобнее)

12. (review) Расскажите о контейнере std::vector. Предложите реализацию методов push_back и pop_back (с правильной поддержкой аллокаторов, move-семантики и гарантий безопасности относительно исключений).

Что такое вектор и его тело, см. в предыдущем пункте.

```
template <class T, class Alloc>
void vector<T, Alloc>::push_back(const T& value) {
    // Если можно просто положить, то всё просто
    if (sz < capacity) {
        traits::construct(alloc, data + sz, value);
        ++sz;
    }
    else if (sz == capacity) {
        size_t new_capacity = capacity * 2;
        // Выделяем новую память в 2 раза больше
        T* new_data = traits::allocate(alloc, new_capacity);
        // Кладём туда value (1)
        traits::construct(alloc, new_data + sz, value);
        //                                      ^
        // На самом деле нужна вторая перегрузка, |
        // принимающая T&& value,                 |
        // и тогда здесь ─────────────────────────┘
        // нужно написать std::move_if_noexcept(value)

        // Ну, и поэлементно перемещаем из старой памяти в новую.
```

```
        for (int i = 0; i < sz; ++i)
            traits::construct(alloc, new_data + i,
std::move_if_noexcept(data[i]));

        destruct();  // Уничтожили старое и деаллоцировали память
        data = new_data;
        capacity = new_capacity;
        ++sz;
    }
}

template <class T, class Alloc>
void vector<T, Alloc>::pop_back() {
    if (sz == 0) return;
    traits::destroy(alloc, data + sz - 1);
    sz -= 1;
}
```

(1) Представьте ситуацию, когда мы делаем так: `v.push_back(v[5]);`
и нам надо перевыделить память. Если бы мы сначала перемещали старые элементы, а потом добавляли бы новый, то к моменту, когда мы хотим добавить новый, он уже может быть невалиден, так как был move'нут. Поэтому мы сначала добавляем его в конец, а уже потом перемещаем остальные.

Дополнения:

1. Опять же, если есть вопросы/исправления, пишите

13. (review) Расскажите про vector<bool>. В чем его особенность и отличие от обычного vector? Расскажите, каким образом достигается эта особенность с точки зрения реализации (опишите реализацию основных методов).

1) Общие слова про особенность:
vector<bool> — отдельная специализация std::vector. Тип данных bool, как мы знаем, занимает 1 байт, хотя его можно уместить в 1 бит. И vector<bool> организован так, чтобы 1 элемент в нем занимал не 1 байт, а в 8 раз меньше, с сохранением стандартной работы всех остальных методов.

2) `operator[]`:
Если мы хотим экономить память, то в момент присваивания должна возвращаться не ссылка на элемент, а элемент некоторого промежуточного класса (назовём `BoolProxy`), который в момент присваивания будет делать "нечто, что изменит вектор" и по смыслу будет как присваивание. Каждый объект данного класса отвечает за свой индекс, и в остальных случаях ведет себя как bool. В грубом приближении описание реализации выглядит следующим образом:

```
template <class Allocator = std::allocator<bool>>
class vector<bool, Allocator>{
public:

   class BoolProxy {
   public:
       //правило пяти

       BoolProxy(unsigned char *ptr, unsigned char bit_offset)
       : ptr(ptr), bit_offset(bit_offset) {}
       // устроен так, что когда мы его
```

```
        // конструируем, меняется элемент вектора по указателю ptr
        // со смещением bit_offset

        //чтобы объект класса в выражениях, где ему не делается
присваивание, всегда корректно конвертировался в bool
        operator bool() const noexcept {
            return (*ptr) & (1 << bit_offset);
        }

        BoolProxy& operator=(bool other) & noexcept {
            if (other)
                (*ptr) |= (1 << bit_offset);
            else
                (*ptr) &= ~(1 << bit_offset);
            return *this;
        }
    private:
        unsigned char *ptr; // Указатель на блок из 8 бит
        unsigned char bit_offset; // Смещение, в битах, влево от правого
конца блока
        // В частности, в блоке 0b00001100 true расположены по смещениям 2 и
3
    };
    // возвращается не ссылка, т.к. мы создаем новый объект
    BoolProxy operator[](unsigned int i) {
        return BoolProxy(buffer_ + i / 8, 7u - i % 8);
    }

    //остальные методы вектора как были

private:
    unsigned char *buffer_;
};
```

3) Отличие от обычного vector:
Строго говоря, vector<bool> — это даже не контейнер STL, и он хранит вовсе не bool (https://cpp.com.ru/meyers/ch2.html#t33, совет 18, кратко пересказан далее). Т.е с vector<bool> нельзя сделать следующее, что можно с обычным вектором:

```
vector<bool> v;

bool *pb = &v[0]; // Инициализировать bool* адресом результата,
возвращаемого оператором vector<bool>::operator[]
```

Будет ошибка компиляции, т.к. `operator[]` возвращает вовсе не ссылку на bool, а ссылку на некоторый `BoolProxy`, и в правой части стоит не указатель на bool
А так как вышеприведённый код не компилируется, то vector<bool> не удовлетворяет требованиям контейнера STL.
Стоит также заметить, что vector<bool> очень медленный из-за сложной внутренней реализации, но при этом экономит память.

14. (review) Расскажите о контейнере std::list. Каковы его методы и скорость их работы? Как он устроен изнутри? Каким образом list<T> использует аллокатор для типа T? Предложите реализацию основных методов list'а: конструкторы, операторы присваивания, деструктор, insert элемента по итератору и erase по итератору.

<list>

1) std::list — двусвязный список.

Основное преимущество — вставка в и удаление из любого места, объединение двух листов за O(1) (гарантированно)

Недостаток — нет `[]` и `add` (что логично, иначе они работали бы за O(n))

2) Основные методы:

- все виды конструкторов для контейнеров
- `push_back, pop_back, push_front, pop_front` — гарантированно за O(1)
- `insert(std::list<T>::const_iterator& it, const T& x)` (вместо `const T& x` можно передавать также временный объект, список инициализации) — вставляет переданный объект на позицию до той, на которую указывает итератор — O(1)
- `insert(std::list<T>::const_iterator& it, InputIter start, InputIter end)` — вставит диапазон от `start` (включительно) до `end` (не включительно) (не обязательно из другого list) на позицию до `it` — O(k), где k — длина диапазона (пересчитываем size)
- `erase(std::list<T>::const_iterator& it)` — удаляет элемент, на который указывает итератор — O(1)
- `erase(const_iterator first, const_iterator last)` — удаляет диапазон — O(k)
- `size()` — возвращает количество элементов в списке — O(1)

3) Специальные методы:

- `splice(const_iterator pos, list& other, const_iterator first, const_iterator last)` — "выпиливает" из переданного листа указанный диапазон элементов и "впиливает" в наш лист без копирований и и перемещений за счёт перестановки указателей — O(n), т.к. мы пересчитываем size, а это занимает линейное время
- `unique()` — удаляет из списка все повторяющиеся элементы
- `sort()` — сортирует элементы списка (`std::sort` не работает, т.к. у `list` нет `operator[]`)
- `reverse()` — переворачивает list — O(n)

4) Проблема аллокаторов:

В листе мы храним некоторую внутреннюю структуру (`Node` или нечто подобное), у которой есть указатель на предыдущий и следующий элементы и также некоторые данные. Нам же передают аллокатор для типа `T`. Чтобы при помощи данного аллокатора выделить `Node`, необходимо использовать следующий метод аллокатора:

```
template <class u>
std::allocator<u> rebind();
```

Данный метод возвращает аллокатор для `u` на основе исходного (точнее, тип нового аллокатора) (см. в list'е). Он реализован в стандартном аллокаторе и должен быть реализован в пользовательских, иначе allocator_traits реализует за нас (с именем `rebind_alloc`), и, возможно, он будет работать некорректно.

~~Уродец из Кунсткамеры~~ реализация list:

https://gist.github.com/VladimirBalditsyn/2cdeaa296c391fd147abada7bbe2dc2d

Советую ознакомиться сначала с реализацией вектора выше, т.к они во многом схожи

Все исправления и замечания пишите мне.

UPD: вопрос: какой typedef должен быть для pointer в итераторе листа?

15. (review) Расскажите о контейнере std::forward_list. Каковы его методы и скорость их работы? Как он устроен изнутри? Каким образом forward_list<T> использует аллокатор для типа T? Предложите реализацию основных методов forward_list'a: конструкторы, операторы присваивания, деструктор, insert элемента по итератору и erase по итератору.

Однонаправленный список. Логично, что все методы работают так же, как и у обычного list, и в реализации аналогичны. Сделаю лишь некоторые замечания:

1) отсутствуют `push_back`, `pop_back`
2) вместо `insert`, `erase`, `splice` — `insert_after`, и т.д.
3) в реализации list нужно в `Node` убрать `prev`, в самом классе убрать `tail`, и произвести соответствующие изменения во всём коде

16. (review) Расскажите о контейнере std::deque. Чем он отличается от vector? Расскажите об адаптерах над контейнерами (std::stack, std::queue, std::priority_queue). Предложите реализацию какого-нибудь одного из них.

std::deque — контейнер для последовательности. Основные отличия от вектора:

- push_back, push_front ровно O(1) — в векторе амортизированно O(1)
- При вставках и удалениях в/из начало/конец не инвалидируются указатели и ссылки на остальные элементы. Пример:

  ```
  vector<int> vec = {1, 2, 3};
  int &ref = vec[1]; // элемент из середины
  vec.push_back(4); // и так 1000 раз
  ```
  С этого момента ref — висячая ссылка, так как, скорее всего, произойдет реаллокация буфера.

  Однако std::deque гарантирует, что после {push,pop}_{back,front} все ссылки и указатели на не-крайние элементы будут валидны.

- Это достигается б**о**льшими накладными расходами памяти

Адаптеры — "обёртки" над другими контейнерами, изменяющие (расширяющие) их API. Шаблонными параметрами принимают тип значения и тип вложенного контейнера.

- std::stack: LIFO (last in, first out): вместо *push_{back,front}* просто *push* на вершину, аналогично pop
- std::queue: FIFO (first in, first out): вставка в хвост, извлечение из головы
- std::priority_queue: очередь с приоритетом, по сути — binary max-heap. *pop* извлекает максимум в контейнере.

Внутренние контейнеры по умолчанию — std::deque для стека и очереди, std::vector для очереди с приоритетом.

Возможная реализация priority_queue:
https://gitlab.com/bcskda/mipt-2sem/blob/master/cpp/stl/priority_queue.hpp

Дополнение: об устройстве std::deque
Идея — https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl
В стандарте в контексте контейнеров STL асимптотика понимается как число *операций над элементами*, поэтому амортизированно-константная вставка в начало/конец массива указателей (*map*) не учитывается в асимптотике => ровно O(1) push_{back,front}.
Код нерабочий, но идея улавливается: https://gitlab.com/bcskda/mipt-2sem/blob/master/cpp/stl/deque.hpp

Дополнение от тов. Титова: можно написать deque как двусвязный список векторов, тогда асимптотика операций будет… лучше (ввиду полного отсутствия амортизации).

17. (done) Расскажите о контейнерах std::map, std::set, std::multimap и std::multiset. Для чего они применяются? Какие у них шаблонные параметры и что они означают? Каковы их основные методы, скорость и принцип работы этих методов? С помощью какой структуры данных реализованы эти контейнеры?

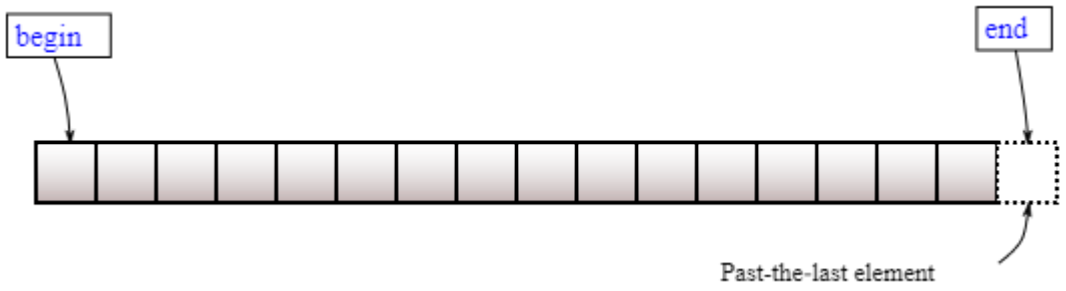
Сразу отметим, что хоть в стандарте и не закреплено, но практически всегда в основе реализации всех 4-ых структур лежит красно-чёрное дерево. Поэтому все методы, вызывающие разбалансировку дерева и оперирующие с одним узлом, работают гарантированно за O(log(n)).

Что же касается принципов работы, по словам лектора, достаточно примерно понимать, как устроены эти методы (дословно про *find()*:"ну, спускаемся по красно-черному дереву, пока не увидим ключ. Работает это по понятным причинам за логарифм"). Так как многие знакомы с основными СД, тут будут отмечены только неочевидные моменты.
Также отметим, что практически у всех функций не проставлены принимаемые/возвращаемые значения, так как это сделает вопрос громоздким. Если кому-то интересно, то сюда.

А далее по порядку.
1) std::map — это отсортированный ассоциативный (т.е. нет понятия первый/последний эл-т[31], но есть понятия ключ и значение, которое ему соответствует) контейнер, который содержит пары ключ-значение с неповторяющимися ключами.
1.1) Фишка map — в порядке (элементы сравниваются по произвольным ключам) и скорости работы (всё строго за O(log(n)), а вот у *std::unordered_map* иногда допускаются операции за O(n) в худшем случае. Но всё же Вы можете спросить, зачем *map* нужен, если есть *unordered_map*. Так вот, компаратор для произвольных пар куда проще реализовать, чем хеш; более того, каждый узел *map* выделяется отдельно, поэтому указатели и итераторы на узел не инвалидируются при работе с остальными узлами (в *unordered_map* может произойти рехеш, и тогда все итераторы инвалидируются).
1.2) Шаблонные параметры:

```
template<class Key, class T, class Compare = std::less<Key>, class
Allocator = std::allocator<std::pair<const Key, T>>
```

Key — тип ключа, T — тип значения, Compare — бинарный предикат сравнения двух ключей (по умолчанию std::less<Key>), Allocator — аллокатор для размещения пар (по умолчанию std::allocator от пары).
1.3) Основные методы std::map:[32]
begin() (cbegin()) возвращает итератор (const итератор) на первый элемент, O(1);
end() (cend()) возвращает итератор (const итератор) на "элемент, следующий за последним", O(1);



[31] Имеется в виду стабильная позиция, ибо есть итераторы и можно рассматривать первым эл-том *mymap.begin()*
[32] Сюда добавлены почти все общие для всех четырёх контейнеров функции.

empty() проверяет отсутствие элементов в контейнере, O(1);
size() возвращает количество элементов в контейнере, O(1);
insert() вставляет эл-ты, O(log(n)), но если вставка происходит сразу после *hint*, то амортизированно O(1); (для диапазона — O(n*log(n+k)), k — кол-во вставленных);
erase() удаляет элементы, O(log(n)) + O(k), где k — либо расстояние (если удаление диапазона), либо число удалённых элементов (если удаление по значению). Сложность такая, так как удаление по итератору — O(1) амортизированно, ребаланс/поиск эл-та — O(log(n)), поэтому из-за хитрого удаления сразу всех эл-тов и всего лишь одного ребаланса получаем данную сложность.
count() возвращает количество элементов, соответствующих определенному ключу, O(log(n)) + O(k), где k — число найденных элементов;
find() находит элемент с конкретным ключом, O(log(n));
lower_bound() возвращает итератор на первый элемент, НЕ меньший, чем заданное значение, O(log(n));
upper_bound() возвращает итератор на первый элемент *больший*, чем определенное значение, O(log(n));
equal_range() возвращает пару из двух итераторов, м/у которыми находятся все равные значения для конкретного ключа, O(log(n)). Тут комментарий схож с комментарием к *erase()*. Поиск ключа выполняется за O(log(n)), инкрементирование итератора амортизированно за O(1) (в худшем случае — за O(log(n))).
merge() сливает два *map,* O(k*log(n+k)), где n и k — размеры map'ов, В который вставляют и ИЗ которого, соответственно.
Также стоит упомянуть две тонкости (вторая только для *map*):
1) метод *insert* также имеет версию с *hint* (т.е. итератор, начиная с которого нужно искать место для вставки), что помогает ускорить вставку;
2) *operator[]* коварный, а именно:
2.1) его нельзя использовать с const map, так как он неконстантный;
2.2) если по данному ключу не было элемента, то он создаст элемент по этому ключу (чтобы не кидать исключение), проинициализирует его по умолчанию и вернёт значение по умолчанию. Если же Вы хотите обращение, кидающее исключение вместо создания, то используйте константный метод *at().*
Пример кода для всех функций см. в конце вопроса, а для *at()* вот:

```
int main() {
    std::map<std::string, int> mymap;

    mymap["a"] = 1;
    std::cout << mymap["a"] << ' '; // prints 1

    try {
        std::cout << mymap.at("b"); // prints "There is no this key"
    } catch (...) {
        std::cout << "There is no this key";
    }
}
```

2.1) std::set — ассоциативный контейнер, который содержит упорядоченный набор уникальных объектов. Далее можно написать всё то же, что и про *map*, ведя разговор только о ключах.
2.2) Фишка set такая же, как и у *map*.
2.3) Шаблонные параметры:

```
template<class Key, class Compare = std::less<Key>, class Allocator =
std::allocator<Key>>
```

Пояснения как и у map.

2.4) Основные методы: те же, что и у map, только нет *operator[]* и *at()*. Чтобы итоговый код не был громоздким, давайте тут приведём пример использования функции *merge* (пытается извлечь каждый элемент из переданного set и вставить их в *this, используя сравнение в смысле *this[33]; не происходит копирования/перемещения node-в, переназначаются только внутренние указатели). Пример кода с merge:

```
int main() {
    std::set<int, std::greater<int>> set1{ 1, 3, 5 };
    std::set<int> set2{ 1, 4, 6 };

    set1.merge(set2);

    for (auto i = set1.cbegin(); i != set1.cend(); ++i) {
        std::cout << *i << ' '; // prints 6 5 4 3 1
    }
    std::cout << '\n';
    for (auto i = set2.cbegin(); i != set2.cend(); ++i) {
        std::cout << *i << ' '; // prints 1
    }
}
```

3.1) std::multimap — это *map*, в котором разрешены пары с одинаковыми ключами (порядок пар с равными ключами определяется порядком вставки и не меняется).
3.2) Фишка multimap такая же, как и у *map* + теперь есть возможность отслеживать диапазоны значений, имеющих одинаковый ключ.
3.3) Шаблонные параметры такие же, как у *map*.
3.4) Основные методы такие же, как у *map*.

3.1) std::multiset — это *set*, в котором разрешены одинаковые ключи.
3.2) Фишка multiset такая же, как и у *set* + теперь есть возможность отслеживать диапазоны одинаковых значений.
3.3) Шаблонные параметры такие же, как и у *set*.
3.4) Основные методы такие же, как у *set*.

Пример кода с использованием всех функций для *std::multimap:*

```
int main() {
    std::multimap<std::string, int> mymap {
        {"first", 1},
        {"first", 2},
        {"first", 3},
        {"second", 4},
        {"second", 5}
    };

    std::multimap<std::string, int>::iterator hint = mymap.find("second");
    mymap.insert(hint, std::make_pair<std::string, int>("third", 6));

    for (auto i = mymap.cbegin(); i != mymap.end(); ++i) {
        std::cout << i->first << ' ' << i->second << '\n';
    }

    auto second_begin = mymap.lower_bound("second"), second_end =
mymap.upper_bound("second");
    mymap.erase(second_begin, second_end);
```

[33] если в *this уже есть такой эл-т, то извлечения не происходит.

```
    std::cout << mymap.empty() << ' ' << mymap.size() << '\n';

    std::cout << "Number of first:  " << mymap.count("first") << '\n';
    auto first_range = mymap.equal_range("first");
    for (auto i = first_range.first; i != first_range.second; ++i) {
        std::cout << i->first << ' ' << i->second << '\n';
    }
}
```

Вывод программы:

```
first 1
first 2
first 3
second 4
second 5
third 6
0 4
Number of first:  3
first 1
first 2
first 3
```

18. (done) Расскажите о контейнерах std::unordered_map, std::unordered_set, std::unordered_multimap и std::unordered_multiset. Для чего они применяются? Какие у них шаблонные параметры и что они означают? Каковы их основные методы, скорость и принцип работы этих методов? С помощью какой структуры данных реализованы эти контейнеры?

Сразу отметим, что в основе реализации всех 4-ых структур лежит хеш-таблица с методом цепочек. Поэтому вставка, поиск и удаление в среднем за O(1 + α), где α — коэффициент заполнения таблицы.

Что же касается принципов работы, по словам лектора, достаточно примерно понимать, как устроены эти методы. Так как многие знакомы с основными СД, тут мы это опустим.

Также отметим, что практически у всех функций не проставлены принимаемые/возвращаемые значения, так как это сделает вопрос громоздким. Если кому-то интересно, то сюда.

А далее по порядку.

1) std::unordered_map — это ассоциативный (т.е. нет понятия первый/последний эл-т, но есть понятия ключ и значение, которое ему соответствует) контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Элементы не отсортированы, но распределены по корзинам (распределение полностью зависит от значения хеша ключа).

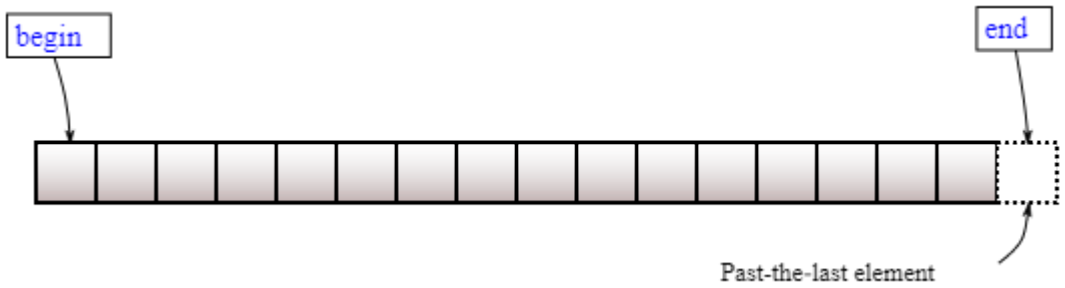
1.1) Фишка unordered_map — нет привязки к сравнению ключей (достаточно определения хорошей хеш-функции), и среднее время работы основных операций - константа, что не может не радовать. Например, очень часто используются, когда нужно быстро вставлять/удалять и проверять существование эл-тов.

1.2) Шаблонные параметры:

```
template<class Key, class T, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator< std::pair<const Key, T> > >
```

Key - тип ключа, T - тип значения, Hash - функция хеширования (по умолчанию std::hash<Key>), KeyEqual - бинарный предикат проверки на равенство двух ключей, Allocator - аллокатор для размещения пар (по умолчанию std::allocator от пары).

1.3) Основные методы std::unordered_map:[34]
begin() (cbegin()) возвращает итератор (const итератор) на первый элемент, O(1);
end() (cend()) возвращает итератор (const итератор) на элемент, следующий за последним, O(1);



empty() проверяет отсутствие элементов в контейнере, O(1);
size() возвращает количество элементов в контейнере, O(1);
insert() вставляет эл-ты, O(1) в среднем (и с hint тоже), O(n) — в худшем, (если диапазон, то в среднем — O(k), в худшем — O(k*n + k));
erase() удаляет элементы, O(k) — в среднем, O(n) — в худшем, где k — либо расстояние (если удаление диапазона), либо число удалённых элементов (если удаление по значению);
count() возвращает количество элементов, соответствующих определенному ключу, O(k) — в среднем (k — число найденных элементов, т.е. константа), O(n) — в худшем;
find() находит элемент с конкретным ключом, O(1) в среднем, O(n) — в худшем;
equal_range() возвращает пару итераторов, м/у которыми находятся все одинаковые значения для конкретного ключа, O(k) — в среднем (k — число найденных элементов, т.е. константа), O(n) — в худшем;
merge() сливает два *unordered_map,* O(k) — в среднем, O(n*k + k), где n и k — размеры map'ов, В который вставляют и ИЗ которого, соответственно.
Следующие функции не являются обязательными к запоминанию, но хорошо держать их в голове:
bucket_count() возвращает текущее число "корзинок" для хранения эл-тов, O(1);
bucket() возвращает индекс "корзинки" для заданного ключа, O(1);
load_factor() возвращает среднее число эл-тов на одну "корзинку", O(1);
max_load_factor() без параметров возвращает load factor, а с переданным float устанавливает новое значение load factor, равное ему, O(1);
rehash(size_type count) делает число "корзин" равным count и запускает процедуру переразмещения значений по "корзинам" (делает rehash), O(n) — в среднем, O(n*n) — в худшем;
reserve(size_type count) устанавливает число "корзинок" равным количеству, необходимому по меньшей мере для размещения count эл-тов без достижения max load factor, и запускает rehash().
Для *unordered_map* стоит отметить, что *operator[]* коварный, а именно:
1) его нельзя использовать с *const unordered_map*, так как он неконстантный;
2) если по данному ключу не было элемента, то он создаст элемент по этому ключу (чтобы не кидать исключение), проинициализирует его по умолчанию и вернёт значение по умолчанию.
Если же Вы хотите обращение, кидающее исключение вместо создания по умолчанию, то используйте константный метод *at()*.
Пример кода для всех функций см. в конце вопроса, а для *at()* вот:

```
int main() {
	std::unordered_map<std::string, int> mymap;

	mymap["a"] = 1;
	std::cout << mymap["a"] << ' '; // prints 1

	try {
		std::cout << mymap.at("b"); // prints "There is no this key"
```

[34] Сюда добавлены почти все общие для всех четырёх контейнеров функции.

```
	}
	catch (...) {
		std::cout << "There is no this key";
	}
}
```

2.1) std::unordered_set — ассоциативный контейнер, который содержит набор уникальных объектов. Далее можно написать всё то же, что и про *unordered_map*, ведя разговор только о ключах.
2.2) Фишка set такая же, как и у *unordered_map*.
2.3) Шаблонные параметры:
`template<class Key, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator<Key>>`
Пояснения как у *unordered_map*.
2.4) Основные методы: те же, что и у map, только нет *operator[]* и *at()*. Чтобы итоговый код не получился перегруженным, приведём пример использования *merge* (пытается извлечь каждый элемент из переданного unordered_set и вставить их в *this, используя хеш-функцию для this и равенство в смысле *this[35]; не происходит копирования/перемещения node-в, переназначаются только внутренние указатели). Пример кода с merge:

```
int main() {
	std::unordered_set<int> set1{ 1, 3, 5 };
	std::unordered_set<int> set2{ 1, 4, 6 };

	set1.merge(set2);

	for (auto i = set1.cbegin(); i != set1.cend(); ++i) {
		std::cout << *i << ' '; // prints 1 3 5 4 6
	}

	std::cout << '\n' << set1.bucket_count() << '\n'; // prints 8
	set1.rehash(12);
	for (auto i = set1.cbegin(); i != set1.cend(); ++i) {
		std::cout << *i << ' '; // prints 1 3 5 4 6
	}
}
```

3.1) std::unordered_multimap — это *unordered_map*, в котором разрешены пары с одинаковыми ключами (порядок пар с равными ключами нестабильный).
3.2) Фишка unordered_multimap такая же, как и у *map* + теперь есть возможность отслеживать диапазоны значений, имеющих одинаковый ключ.
3.3) Шаблонные параметры такие же, как и у *unordered_map*.
3.4) Основные методы такие же, как у *unordered_map*, кроме *operator[]* и *at()*.

3.1) std::unordered_multiset — это *unordered_set*, в котором разрешены одинаковые ключи.
3.2) Фишка unordered_multiset такая же, как и у *unordered_set* + теперь есть возможность отслеживать диапазоны одинаковых значений.
3.3) Шаблонные параметры такие же, как у *unordered_set*.
3.4) Основные методы такие же, как у *unordered_set*.

Пример кода с использованием почти всех функций для std::unordered_multimap:

```
int main() {
	std::unordered_multimap<std::string, int> mymap{
```

[35] если в *this уже есть такой эл-т, то извлечения не происходит.

```
            {"first", 1},
            {"first", 2},
            {"first", 3},
            {"second", 4},
            {"second", 5}
      };

      std::unordered_multimap<std::string, int>::iterator hint =
mymap.find("second");
      mymap.insert(hint, std::make_pair<std::string, int>("third", 6));

      for (auto i = mymap.cbegin(); i != mymap.end(); ++i) {
            std::cout << i->first << ' ' << i->second << '\n';
      }

      auto equal_first = mymap.equal_range("first");
      mymap.erase(equal_first.first, equal_first.second);
      std::cout << mymap.empty() << ' ' << mymap.size() << '\n';
      std::cout << "Number of first:  " << mymap.count("first") << '\n';

      for (auto i = mymap.cbegin(); i != mymap.end(); ++i) {
            std::cout << i->first << ' ' << i->second << '\n';
      }
}
```

Программа выведет:

```
first 1
first 2
first 3
second 4
second 5
third 6
0 3
Number of first:  0
second 4
second 5
third 6
```

19. (review) Что такое итераторы? Какие виды итераторов бывают и какие операции допустимы над каждым из них? Расскажите про функции std::advance и std::distance. Каким образом достигается их разное поведение в зависимости от вида переданного итератора?

1) Итератор — сущность, позволяющая выполнять обход элементов контейнера. Для итератора должно быть определено понятие “следующий элемент” — т.е. к каждому итератору применим пре-инкремент, и доступ к текущему элементу — т.е. разыменовывание.

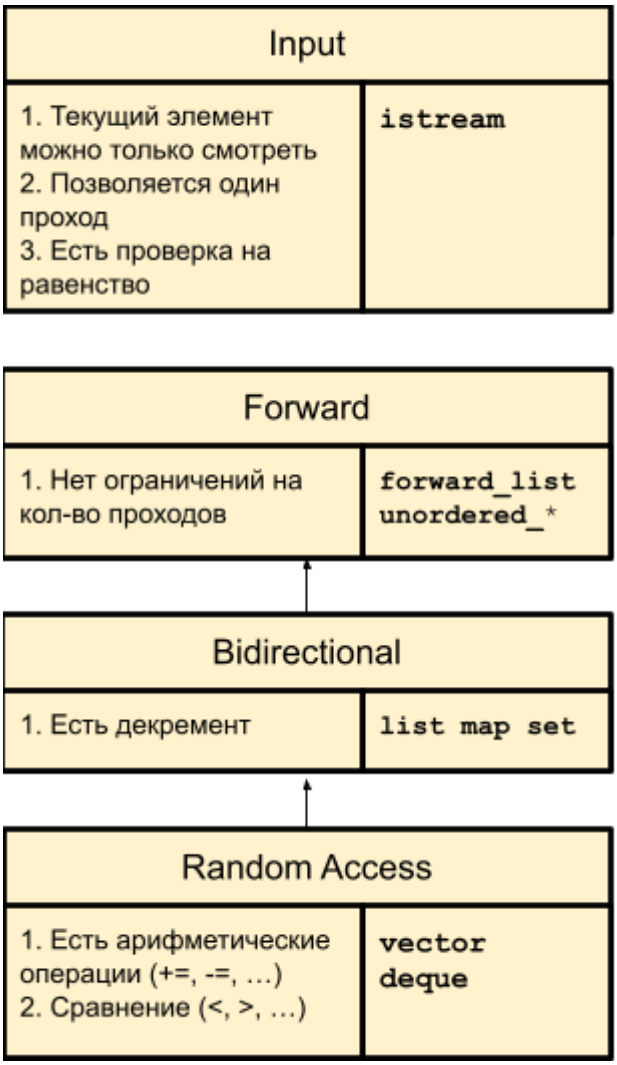
★ *Все итераторы поддерживают* `++` *и* `*`

Так как почти каждый стандартный контейнер предоставляет особенный обход своих элементов, внутри большинства STL-ных контейнеров есть структура `iterator`, позволяющая последовательно проходиться по элементам. Например, так можно создать итератор по map:

```
std::map<int, int>::iterator it;
```

2) Виды итераторов.

2.1) В таблице (стрелки обозначают подмножества) указаны: сверху — название итератора, слева — добавочный функционал, справа — примеры контейнеров (потоков) с соответствующими итераторами.



Элемент под Output iterator можно изменить 1 раз (не обязательно можно читать) (после чего итератор должен быть инкрементируем), то есть позволяет только один проход

Итератор может быть одновременно input и output — это независимые понятия.
2.2) STL-контейнеры предоставляют методы:
begin() — возвращает итератор на начало контейнера
end() — на конец контейнера
Т.е. к примеру, такой конструкцией можно пройтись по всем элементам сета:

```
std::set<int> set;
for (std::set<int>::iterator it = set.begin(); it != set.end(); ++it) {}
```

Однако контейнер может дать доступ только на чтение элементов (т.е. по нему можно проходиться только Input iterator-ом). В таком случае, чтобы сказать, что мы не будем изменять элементы контейнера, есть `const_iterator` и методы cbegin() и cend(), возвращающие уже константные итераторы:

```
for (std::set<int>::const_iterator it = set.cbegin(); it != set.cend();
++it) { *it += 1; // error }
```

Также есть `reverse_iterator` (разумеется, для контейнеров, поддерживающих Bidirectional iterator-ы), позволяющий пройтись с конца к началу и соответствующие rbegin() и rend():

```
for (std::set<int>::reverse_iterator it = set.rbegin(); it != set.rend();
++it) {}
```

Есть и `const_reverse_iterator`, сочетающий предыдущие два.

3) Ранее (до C++17) итераторы предлагалось наследовать от `std::iterator` (заголовочный файл <iterator>), в котором было много typedef-ов (и больше ничего). При создании собственного итератора при наследовании надо было указывать в шаблоне “тег”, т.е. делать пометку, какой вид имеет итератор из приведённых выше 5 (таблица). Делалось это как-то так:

```
class insert_iterator
: public iterator<output_iterator_tag, void, void, void, void>
```

Однако сейчас это ушло в прошлое, а на замену пришёл `std::iterator_traits`, который также содержит typedef-ы, но делает это более умно — **например, сам определяет тег переданного итератора**[36] [37] [38].
Для того чтобы различать виды итераторов в языке есть соответствующие классы:

input_iterator_tag output_iterator_tag forward_iterator_tag bidirectional_iterator_tag random_access_iterator_tag	пустые классы, используемые для обозначения категорий итераторов (класс)

Собственно в `std::iterator_traits<...>::iterator_category` лежит один из этих типов. Все функции, использующие итераторы — когда хотят что-то узнать об итераторе — обязаны обращаться к iterator_traits, т.к. в этой структуре реализованы методы для удобной работы с итераторами. К примеру, чтобы узнать тег итератора типа Iterator, можно использовать `std::iterator_traits<Iterator>::iterator_category`.
Для иллюстрации использования iterator_traits реализуем `std::distance` и `std::advance`.

```
template<typename Iterator>
size_t distance(Iterator first, Iterator last)
```

Это функция, которая возвращает количество шагов от first до last. Она должна работать эффективно, а это значит, что если на вход подаётся RandomAccessIterator, то вместо пошагового инкремента (который применяется для всех остальных видов итераторов), функция должна сделать вычитание. Это достигается путём шаблонной специализации.
Реализация distance:
https://gist.github.com/Wutem/2cffe703c0e1b533b2ff8493c95515c7

```
template<typename Iterator>
Iterator advance(Iterator iterator, size_t n);
```

Продвигает итератор на заданное расстояние. Реализована аналогично distance. (Оставляется в качестве упражнения читателю)

Дополнения:

1. Стоит упомянуть, что есть более общий класс `std::reverse_iterator`, частными случаями которого являются reverse_iterator-ы в контейнерах.
   Реализация (Илья говорил, что могут дать как задачку):
   https://gist.github.com/Wutem/51a5f6bea173b2548dc17004fb3adea0

---

[36] Делает он это, как можно догадаться, через SFINAE: проверяет наличие соответствующих методов для определения вида итератора.
[37] ВАЖНО! Это работает только с C++20! Пока что надо самому определять эти typedef’ы в стиле:
`typedef std::random_access_iterator_tag iterator_category;`
[38] Да, старый функционал уже помечен устаревшим, а нового ещё не подвезли, гениально, согласен

20. (done) Какие виды итераторов поддерживает каждый из стандартных контейнеров? Предложите реализацию итераторов для какого-нибудь из стандартных контейнеров (на ваш выбор), считая, что в остальном контейнер уже реализован.

1) Виды итераторов: см. таблицу в предыдущем билете.
2) Реализация итератора для `std::vector` (точнее для предложенного в 11 билете вектора): https://gist.github.com/Wutem/20d76841c664cbb01443530d19b85db6
Примечание: **в остальном контейнер уже реализован** — кажется, имеется в виду, что не надо реализовывать даже методы контейнера, связанные с итераторами, так что они опущены. Однако в полной версии vector (из 11 билета) с итераторами есть методы begin() и end() для примера.

21. (review) Расскажите про класс std::insert_iterator и функцию std::inserter, предложите их реализацию.

1) Представим, что нам нужно скопировать один вектор в конец другого. Для этого, как известно, существует удобная функция `std::copy(first, last, other_container_first)`, объявленная в <algorithm>. Мы могли бы написать что-то вроде:

```
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = {0};
std::copy(v1.begin(), v1.end(), v2.end());
```

Однако это не даст желаемого результата, т.к. под v2.end и далее лежит (точнее может лежать, если вектор не аллоцировал больше памяти) чужая память и запись в неё есть UB.
Решить проблему помогает класс, конструщийся от контейнера (и итератора — начальной позиции) и являющийся “обёрткой” над итератором — `std::insert_iterator`.

```
template<class Container>
class insert_iterator;
```

Он перехватывает инкремент и разыменовывание (они, кстати говоря, ничего не делают), а вот присваивание вызывает insert у контейнера, от которого insert_iterator сконструировался, и инкрементирует обёрнутый итератор.
Таким образом, следующий код сделает то что нужно:

```
std::copy(v1.begin(), v1.end(), std::insert_iterator<std::vector<int>>(v2,
v2.end()));
//v2 = {0, 1, 2, 3};
```

Можно использовать более удобную функцию `std::inserter` (принимает контейнер и указатель на место вставки), которая за нас выводит шаблонный параметр и возвращает insert_iterator. Т.е. код можно переписать:

```
std::copy(v1.begin(), v1.end(), std::inserter(v2, v2.end()));
```

2) Реализации std::insert_iterator и std::inserter:
https://gist.github.com/Wutem/d4cc9765a6279b6bf79a55041af7df17

ЧАСТЬ 2

22. (review) Расскажите об операторах new и delete. Зачем они нужны, как ими пользоваться? Что такое placement new? В чем разница между оператором new и функцией operator new? Что из вышеперечисленного можно перегружать и как это делать? Зачем может быть нужно перегружать кастомный operator delete? Расскажите о проблеме утечек памяти.

1) Операторы *new* и *delete* необходимы для создания и удаления объектов в динамической памяти [39] Оператор *new* запрашивает у операционной системы необходимое количество памяти (равное[40] размеру объекта) и создаёт на этом месте объект с помощью конструктора. Оператор *delete* разрушает объект с помощью деструктора, потом возвращает операционной системе ранее занятую память. *new* возвращает указатель (соответствующего типа) на только что созданный объект, *delete* возвращает void.
Базовый синтаксис:

```
int *p = new int(5);
delete p;
T *pp = new T(5, 0);
delete pp;
```

Также есть операторы *new[]* и *delete[].* Они нужны для создания c-style массивов (по умолчанию тоже в динамической памяти), которые, по сути, являются непрерывным отрезком памяти размера N*sizeof, и указателя на начало промежутка. Это совсем другие операторы, нельзя удалять то, что создано *new*, через *delete[]* и наоборот[41].

```
int *a = new int[50];
delete[] a;
```

При создании C-style массива можно пользоваться только конструктором по умолчанию:

```
T *p = new T[5];
```

при отсутствии конструктора по умолчанию произойдёт CE.
Hack:
Но с помощью placement-new (см. далее) можно сделать и свой конструктор:

```
int N = 10;
char* pp = new char[N*sizeof(T)];
for (int i = 0; i < N; ++i) {
   new(pp+sizeof(T)*i) T(/* T constructor arguments */);
}
```

(хитро, да? Предложено тов. Купцовым)

Все фишки далее находятся в <new>.
2) Теперь о *placement-new*. Можно вызвать оператор *new* так, чтобы он не выделял новый участок памяти, а создавал объект на уже выделенном участке, на который у нас есть указатель. Это делается так: в качестве аргумента *new* передаётся void*

```
int *a = new int(5);
new(a) T(2, 3);
```

или

```
int *a = new int(5);
T *p = new(a) T(2, 3);
```

(Таким образом можно вызывать *new* и на стек тоже).
Кстати, если T меньше int, то в этих примерах у вас будет UB

3) Что вообще делает оператор new? Он состоит из двух частей: сперва вызывается функция `operator new`, которая, собственно, выделяет память, а затем вызывается конструктор на это место выделенной памяти. Первую часть можно перегрузить (глобально или для каждого класса).
Синтаксис (здесь и далее я перегружаю operator new для класса):

```
static void *operator new(size_t n) {
   return ::operator new(n); //перенаправление на стандартный operator new
как затычка
```

---

[39] placement-new позволяет вызывать new на любом участке памяти, так что говорить, что new нужен для работы с динамической памятью, не вполне корректно, oh well
[40] Есть подозрение, что new выделяет больше памяти — у меня ошибку сегментации не бросало при попытке сконструировать б**о**льший объект, кидало только при значительном сдвиге
[41] Будет UB, см. объяснение к (1) и (2) https://en.cppreference.com/w/cpp/memory/new/operator_delete

```
}
```

Почему static, спросите вы? Вообще это логично, но и без static будет тоже работать[42]. Здесь `n` — количество байт, которое надо выделить. new вызывает operator new, неявно передавая туда sizeof от создаваемого объекта. Также можно переопределить placement-new:

```
static void *operator new(size_t n, void *p) {
   return ::operator new(n, p); //перенаправление на стандартный operator
new
}
```

Кстати, стандартный placement-new, как можно догадаться, выглядит так:

```
void *operator new(size_t n, void *p) {
   return p;
}
```

Вообще, можно создавать operator new с каким угодно количеством аргументов:

```
static void *operator new(size_t n, void *p, char cool_char, int
important_int) {
   return …;
}
```

а потом вызывать new таким образом:

```
T* p = new(a, 'b', 42) T(2, 3);
```

то есть передавать как аргументы new, начиная со второго.

Зачем это нужно? Для какой-то хитрой аллокации (или хотя бы для логирования)

4) delete работает аналогично: сперва вызывается деструктор, потом вызывается `operator delete`

```
static void operator delete(void *p, size_t n) {[43]
   ::operator delete(p, n);
}
```

Опять таки можно создать operator delete со многими переменными, нельзя вызвать delete от многих переменных как new — нет такого синтаксиса

```
static void operator delete(void *p, size_t n, char cool_char) {
   ::operator delete(p, n);
}
```

…

```
delete('d') p; //нельзя
```

Надо явно сперва вызвать деструктор, а потом operator delete

```
p->~T();
T::operator delete(p, sizeof(*p), 'c');
```

Зачем это нужно? Если у нас какая-то хитрая аллокация, нам может понадобиться какая-то хитрая деаллокация.

Настоятельно рекомендуется в случае, если вы написали кастомный operator new, написать operator delete с точно такими же параметрами. И вот почему: пусть конструктор кидает исключение. Но operator new уже отработал, и память выделилась, но из-за исключения сами мы её не освободим. В этой *исключительной* ситуации обязан вызваться operator delete с такими же параметрами, что и operator new. Если такой не нашлось, можно получить утечку памяти.

5) Об утечках памяти. Если мы выделим память в куче, а потом её не освободим (указатель выйдет из области видимости, например), она останется занятой до конца работы программы (так как ОС не будет знать, что мы закончили с этой памятью работать). Это называется *утечкой памяти*. Количество таких “брошенных” кусков памяти может неконтролируемо увеличиваться, и в

---

[42] Из https://en.cppreference.com/w/cpp/memory/new/operator_new#Class-specific_overloads: The keyword static is optional for these functions: whether used or not, the allocation function is a static member function.

[43] Почему-то передавать размер удаляемого объекта необязательно, можно просто передать указатель, но разница слишком тонкая чтобы с ней возиться

итоге вся память будет занята, что вызовет много проблем (в частности, оператор new кинет исключение std::bad_alloc)

Ещё немного об operator new.
6) nothrow new
Чтобы вызвать operator new который не кидает bad_alloc, нужно написать:

```
T* p = new (std::nothrow) T(2,3);
```

Возвращает nullptr в случае ошибки. Его тоже, конечно, можно переопределить, но за подробностями реализации читай <new> (там ничего сложного, но переписывать сюда явно незачем).

7) new_handler
new_handler — функция, которая пытается исправить нехватку памяти
set_new_handler принимает указатель на некую функцию
get_new_handler возвращает указатель на эту функцию — или nullptr, если эта функция не определена
new, прежде чем бросить bad_alloc, в бесконечном цикле вызывает get_new_handler. Если он вернул nullptr, то new кидает bad_alloc. Иначе вызывает функцию, которую мы передали в set_new_handler, и после этого пытается ещё раз выделить память. Если не получается, то снова вызывает get_new_handler.
Перегруженные operator new должны вести себя хорошо и действительно в бесконечном цикле вызывать get_new_handler. А ещё при попытке сконструировать 0 байт тоже должен возвращаться корректный указатель.

8) Наследование
Какую версию operator new вызвать, решается динамически (чуть ли не единственный случай, когда вызов статического метода решается динамически)

```
Base *b = new Derived(1, 2);
delete b;
```

вызовется operator new из Derived, шок

23. (review) Что такое аллокаторы, зачем они нужны? Расскажите про класс std::allocator, предложите его реализацию.
1) *Аллокатор* — объект, инкапсулирующий в себе выделение/освобождение памяти и конструирование/разрушение объектов в этом участке памяти. Они нужны для сокрытия вызовов new и delete (так как new и delete — плохие операторы, от вызова которых обычно возникает куча проблем, плюс new выделяет память не особо эффективно). В коде аллокатора, по сути, будет то, что мы написали бы в перегруженном operator new и operator delete.
2) Стандартный аллокатор: просто обёртка над вызовами new и delete, которая не делает ничего дополнительного. Примерная реализация стандартного аллокатора:

```
//Пример реализации стандартного аллокатора
template<typename T>
class MyAllocator {
public:
   //выделить память под n штук типа T
   T *allocate(size_t n) const;
   //освободить память из под n штук
   void deallocate(T *p, size_t n) const;
   //сконструировать объект на некотором участке памяти
   template<typename... Args>
   void construct(T *p, Args &&... args) const;
   //разрушить объект
   void destroy(T *p) const;
```

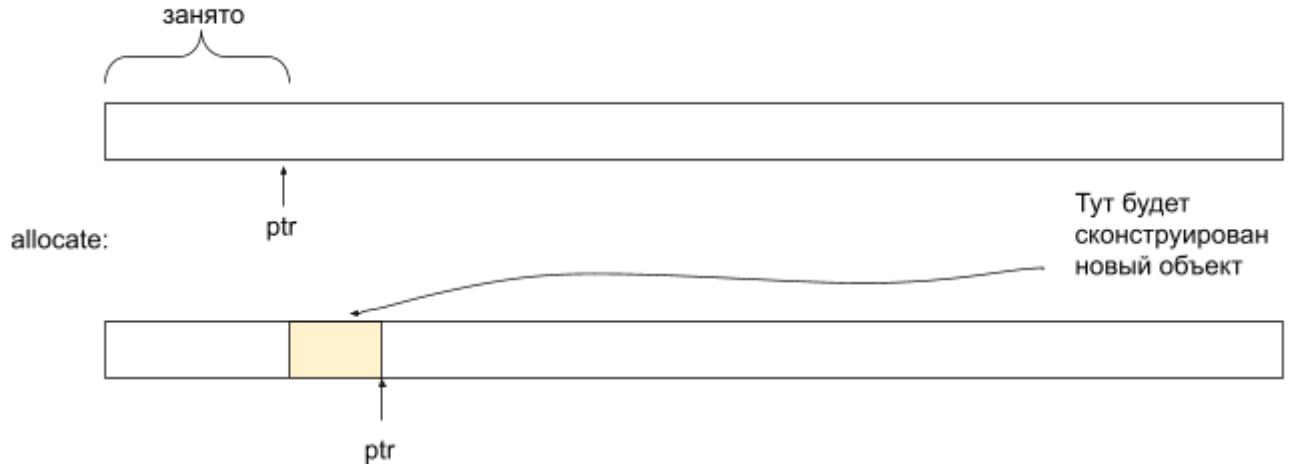
```
};
template<typename T>
T *MyAllocator<T>::allocate(size_t n) const {
   return static_cast<T*>(::operator new(sizeof(T) * n)); //выделение куска
для n штук объектов T
}
template<typename T>
void MyAllocator<T>::deallocate(T *p, size_t n) const {
   ::operator delete(p, n);
}
template<typename T>
template<typename... Args>
void MyAllocator<T>::construct(T *p, Args &&... args) const {
   ::new(p) T(std::forward<Args>(args)...);
}
template<typename T>
void MyAllocator<T>::destroy(T *p) const {
   p->~T();
}
```

В действительности, у всех аллокаторов construct и destroy выглядят одинаково, поэтому в C++20 у аллокаторов их больше не будет.
construct и destroy следует вызывать через:

3) std::allocator_traits, который вызывает соответствующую функцию у аллокатора, если она у него реализована, или стандартную реализацию иначе (std::allocator_traits ~~довольно просто~~ очень сложно реализован через SFINAE).
Вызов construct и allocate через allocator_traits выглядит примерно так:

```
MyAlloc<int> a;
int* ptr = std::allocator_traits<MyAlloc<int>>::allocate(a, 5);
std::allocator_traits<MyAlloc<int>>::construct(a, ptr, 4);
std::allocator_traits<MyAlloc<int>>::construct(a, ptr + 2, 50);
std::cout << ptr[0] << "\n";  // Выведет 4
std::cout << ptr[2];          // Выведет 50
// Прим: должен быть в аллокаторе typedef T value_type
```

4) Пример нестандартного аллокатора
Stack/Pool/... Allocator — аллокатор, который сразу при создании выделяет большой кусок памяти, в allocate он возвращает указатель на последнюю свободную ячейку и сдвигает этот указатель, deallocate не делает ничего (точнее, сдвигает указатель влево, если удаляемая ячейка — последняя). tl;dr — выделяет память stack-style. Такая стратегия выделения позволяет сильно сэкономить время, так как выделение памяти — зачастую долгая операция, а так мы запрашиваем память у ОС всего один раз, а потом сами распоряжаемся ей, как хотим.



Можно придумать и более сложные стратегии выделения памяти, это зависит от ваших нужд.

5) А как копировать аллокатор? Можно делать глубокую копию, то есть перевыделять память. Можно не перевыделять память, а копировать указатель на один и тот же участок памяти, но тогда придётся хранить счётчик указывающих на эту память аллокаторов, чтобы не делать многократный delete по одному месту. shared_ptr, наверное, решает эту проблему? хз[44]. Особенно остро проблема с копированием возникнет в теме по контейнерам.

24. (review) Расскажите о проблемах, для решения которых была введена move-семантика. Расскажите, как работает функция std::move и как ее применять, объясните, как она реализована и почему именно так. Как правильно реализовать функцию swap в C++11?

До C++0x был лишь один способ передавать объекты в другую область памяти — полное копирование (ссылки не считаются, так как ссылка — просто другое имя для той же области памяти). Зачастую полное копирование — очень дорогая операция, так как для сложных объектов придётся переаллоцировать много больших участков памяти, а потом полностью переписывать туда данные (представьте, например, полное копирование вектора векторов).

Можно ли как-то сэкономить на выделении памяти? Можно, если мы знаем, что экземпляр объекта, лежащий под старым именем, нам больше не пригодится. Тогда вот как мы поступим: данные на стеке мы скопируем как обычно, а данные в куче мы не будем копировать, а просто скопируем указатели на эту память. Но если мы так сделаем, память в итоге очиститься дважды, что приведёт к ошибке сегментации! И тут мы вспоминаем, что старый объект-то нам не нужен, значит мы никогда не будет разыменовывать этот указатель, значит его можно просто обнулить, и двойного очищения памяти не будет! Можно на всякий случай обнулить все поля, чтобы сделать объект истинно нулевым.

Такое оптимизированное копирование называется *перемещением*.

Красивую картинку и код, иллюстрирующие разницу между копированием и перемещением, смотри в теорминимуме

Теперь посмотрим, можем ли мы реализовать перемещение средствами C++03 (спойлер: нет). Какие объекты мы хотим перемещать? Прежде всего, временные объекты — они нам явно не пригодятся после того, как мы их переместили в нужную область памяти[45]. Также мы можем

[44] Kinda
[45] Тут надо быть аккуратным так как достаточно часто вместо перемещения будет работать copy elision, RVO и NRVO, смотри далее

захотеть переместить объекты в постоянной памяти. При этом нам нужно иметь возможность изменять старый объект, чтобы занулить его, значит, его надо передавать по ссылке. Но type& не может привязываться к временным объектам, а const type& нельзя изменять[46].
Значит, нам нужен новый вид ссылок — *rvalue-ссылки*. Они могут привязываться и к временным, и к постоянным объектам (перед этим их надо привести к “временным” объектам, об этом позже), и объект под ними можно изменять.

Итог: rvalue-ссылки и все смежные инструменты введены в язык для более эффективного копирования объектов — перемещения.

Несколько стандартных примеров “грустных историй”, которые возникают без семантики для операций перемещения:

```
void swap(String& a, String& b){
   String t = a;
   a = b;
   b = t;
}
```

Произойдёт аж 3 копирования. Но можно же сделать 3 перемещения! (очевидно почему)

```
std::vector<String> v;
String s("abracadabra",6);
//потенциальная работа со строкой
v.push_back(s);
```

Строка s создавалась исключительно для запихивания в вектор, дальше она не нужна. Дешевле просто переместить в вектор

```
String foo(){
   return {"foo", 3};
}
…
String s("abracadabra",6);
s = foo();
```

Пусть мы хотим присвоить объекту временный объект — тот, который только что вернулся функцией. Но вызовется конструктор копирования, что неэффективно по времени.

Зачем нужна функция std::move? Если неформально: она помечает постоянный объект как временный, то есть как объект, который можно переместить (читай: испортить). Реализация std::move:

```
template <typename T>
std::remove_reference_t<T>&& move(T&& a){
   return static_cast<std::remove_reference_t<T>&&>(a);
}
```

tl;dr: эта функция — сокращение к касту к rvalue-ссылке[47]. *По правилам языка* static_cast к type&& возвращает xvalue — постоянный объект, который ведёт себя как временный и который можно будет передать в конструктор перемещения (см. далее)

---

[46] И вообще, const type& уже используется для конструктора и оператора копирования
[47] Многие думают, что move уже портит объект. Это не так, move всего лишь помечает объект перемещаемым, то есть выберется конструктор перемещения, который уже попортит объект

Про возвращаемое значение: std::remove_reference_t<T> мы снимаем все амперсанды, которые были у T, затем через && мы навешиваем обратно два амперсанда. Таким образом, std::remove_reference_t<T>&& — rvalue-ссылка, которая *по правилам языка* необходима для того, чтобы функция вернула xvalue от каста.

Про принимаемое значение: что такое T&&? Это костыль, который неофициально называется *универсальной ссылкой (официально — forwarding reference, формально (но не совсем точно) — rvalue-ссылки в контексте вывода типов*[48]*)*. Эта шаблонная конструкция разворачивается в ссылку с правильным количеством амперсандов, а именно: если передать lvalue типа U, то вместо T будет подставлен U&, если rvalue, то U[49] (далее срабатывает *сжатие ссылок*). Подробнее про универсальные ссылки и сжатие ссылок вы можете прочитать в одном из следующих билетов.

Корректная реализация std::swap:

```
template <typename T>
void swap(T& a, T& b){
   T c = std::move(a);
   a = std::move(b);
   b = std::move(c);
}
```

25. (review) Расскажите о том, что такое “идеальная передача” (perfect forwarding). Расскажите, как работает функция std::forward и как она реализована. Приведите пример применения этой функции. Почему в std::forward нельзя принять T&& в качестве параметра?

perfect forwarding aka идеальная передача aka прямая передача — передача переменных, сохраняя их категорию значения (rvalue или lvalue). Прямая передача реализована с помощью `std::forward`. Это функция, которая возвращает xvalue (кастует к type&&), если в неё передали rvalue-ссылку или rvalue, и lvalue-ссылку, если в неё передали lvalue. То есть она работает как std::move, но кастует она тогда и только тогда, когда объект уже временный, или постоянный, помеченным временным, а постоянные объекты она не трогает.

Зачем это нужно? Вспомним make_unique, который будет через несколько билетов:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args &&... args) {
   return unique_ptr(new T(std::forward<Args>(args)...));
}
```

Та же проблема в construct у аллокаторов:

```
template<typename T>
template<typename... Args>
void MyAllocator<T>::construct(T *p, Args &&... args) const {
   ::new(p) T(std::forward<Args>(args)...);
}
```

Мы ничего не знаем о параметрах Args — какие из них lvalue-ссылки, какие rvalue-ссылки. Если передавать просто args, то type&& в конструкторе не сможет привязаться к lvalue args. Если передавать через std::move, то он приведёт к xvalue и то, что было передано по lvalue ссылке, и

---

[48] Да-да, это также касается auto&&. Точное определение

[49] Я не очень понимаю, зачем вы бы хотели передавать rvalue в std::move, но почему бы и нет

type& в конструкторе не сможет привязаться к xvalue. Поэтому нам нужно на часть делать move, а на часть — нет. std::forward так и работает.

Это также актуально, если мы получаем в функцию-обёртку универсальную ссылку, а внутренняя функция имеет перегрузки и для lvalue-, и для rvalue-ссылок.

Скотт Мейерс[50] рекомендует использовать std::forward для универсальных ссылок, а std::move — для rvalue-ссылок.

Реализация:

```
template <typename T>
T&& forward(std::remove_reference_t<T>& a){
   return static_cast<T&&>(a);
}

template <typename T>
T&& forward(std::remove_reference_t<T>&& a){
   return static_cast<T&&>(a);
}
```

Про возвращаемое значение: если объект был lvalue, то вернётся lvalue-ссылка, если объект был rvalue-ссылка или rvalue (для последнего используется специальная перегрузка), то он кастуется к xvalue — это достигается с помощью правила *сжатия ссылок* (см. билет 27).

Про принимаемое значение:
1) Если передаётся rvalue-ссылка или другие lvalue, то принимаемое значение становится type& — вызывается первая реализация
2) Если передаётся rvalue, то принимаемое значение превращается в type&& — вызывается вторая реализация, специальная для rvalue. Мещерин, кстати, эту реализацию не упоминал: хотя она есть в стандарте, она совершенно бесполезная — зачем бы в forward понадобилось передавать rvalue?

Still confused? Давайте разберём каждый случай вывода шаблонов.
Пусть у нас есть такой код:

```
template<typename T>
std::remove_reference_t<T>&& move(T&& a) {
   return static_cast<std::remove_reference_t<T> &&>(a);
}

template<typename T> //(1)
T&& forward(std::remove_reference_t<T>& a) {
   return static_cast<T &&>(a);
}

template<typename T> //(2)
T&& forward(std::remove_reference_t<T>&& a) {
   return static_cast<T &&>(a);
}

template<class T>
void foo(T&& x) {
   forward<T>(x);
```

[50] Количество букв ‘е’ в фамилии не оспаривается, и так сойдёт

```
}
```

1) Пусть в foo передаётся lvalue:

```
int main() {
   int a = 5;
   foo(a);
}
```

или

```
int main() {
   int&& a = 5;
   foo(a);
}
```

T в foo выводится как int& по правилам универсальной ссылки, следовательно, по правилу сворачивания ссылок (int&) && → int&, T&& выводится как int&. int& далее подставляется явно как T в forward. Выбирается первая реализация forward, которая принимает std::remove_reference_t<int&>& → int&. В универсальную ссылку в касте и в возвращаемом значении подставляется (int&) && → int&, каст происходит к lvalue-ссылке, то есть ничего не происходит, и forward возвращает lvalue-ссылку.

2) Пусть в foo передаётся rvalue:

```
int main() {
   int a = 5;
   foo(move(a));
}
```

или

```
int main() {
   foo(5);
}
```

T в foo выводится как int по правилам универсальной ссылки, поэтому T&& выводится в int&& ((int) && → int&&). int далее подставляется явно как T в forward. Опять таки выбирается первая реализация forward, так как std::remove_reference_t<int>& → int&. В универсальную ссылку подставляется int, поэтому T&& превращается в int&&. Происходит каст к rvalue-ссылке, а превращается в xvalue. forward возвращает xvalue

3) Особо упоротый случай, когда в forward оправляется rvalue:

```
int main() {
   forward<int>(5);
}
```

или

```
int main() {
   int a = 5;
   forward<int>(move(a));
}
```

Выбирается вторая реализация forward, так как std::remove_reference_t<int>& → int& из первой реализации не может привязаться к rvalue, а std::remove_reference_t<int>&& → int&& — может (как раз для этого случая и была введена вторая реализация). Далее аналогично второму пункту.

Почему forward не может принимать универсальную ссылку?

Не знаю, давайте попробуем, авось сработает

```
template<typename T>
```

```
T&& cool_and_new_forward(T&& a) {
   return static_cast<T &&>(a);
}

template<class T>
void foo(T&& x) {
   cool_and_new_forward<T>(x);
}
```

Опять разберём случаи.

Случай, когда мы отдаём lvalue в foo, cool_and_new_forward работает аналогично ~~lame_and_old_~~forward. Но что произойдёт, если в foo отдать rvalue?

```
int main() {
   int a = 5;
   foo(move(a));
}
```

или

```
int main() {
   foo(5);
}
```

T в foo выведется в int, поэтому T&& выведется в int&&. Значит, T&& в принимаемом значении cool_and_new_forward превратится в int&&, а эта ссылка не сможет привязаться к x, который lvalue так как он — идентификатор. Не скомпилируется!

Но погодите, скажете вы, у нас же теперь в принимаемом значении нет std::remove_reference_t, значит, тип может сам вывестись, и его не надо явно указывать. Тогда, вроде, универсальная ссылка в принимаемом значении cool_and_new_forward должна правильно раскрываться. Давайте и этот вариант попробуем:

```
template<typename T>
T&& cool_and_new_forward(T&& a) {
   return static_cast<T &&>(a);
}

template<class T>
void foo(T&& x) {
   cool_and_new_forward(x);
}
```

Но тут мы обламываемся ещё веселее — вне зависимости от того, передаём мы в foo rvalue или lvalue, T в cool_and_new_forward будет всегда выводиться как int& (чтобы принять x, который lvalue, T&& → T&, то есть, (int&) && → int&), и каста к xvalue не будет происходить никогда.

Не передавая шаблонный параметр в forward явно, мы теряем информацию о категории значения T, и в forward мы её никак восстановить не можем. Таким образом, мы доказали, что реализация std::forward должна быть такой и никакой иначе. ∎

That’s it. Let it sink in for a minute. *gurgle*

26. (review) Расскажите об rvalue-ссылках и об универсальных ссылках. Какие присваивания между lvalue-, rvalue-ссылками, а также временными объектами и именованными не ссылочными объектами разрешены, а какие запрещены? Что меняется в случае с константными ссылками и объектами? Что является и что не является универсальными ссылками? Приведите примеры, иллюстрирующие все вышесказанное.

Если обычные lvalue-ссылки привязываются к не временным объектам (const ссылка — к временным), то rvalue-ссылки привязываются к временным объектам (или к не временным, если сделать их xvalue).
Сразу развею распространённое заблуждение: **rvalue-ссылка — не rvalue.** Идентификатор с типом **“**rvalue-ссылка” — такое же lvalue, как и любой другой идентификатор[51]. Каст к rvalue-ссылке — xvalue, да, но только в этом контексте — как только у rvalue-ссылки появляется имя, она перестаёт быть rvalue.

<лажа>
Вы можете, конечно, спросить: падажжи, а где вообще хранится rvalue? Разве оно не хранится в какой-то супер-временной памяти, куда мы даже не можем добраться (там, адрес не можем взять, все дела)? Как тогда мы линкуем ссылку на временный объект, чтобы он стал постоянным? Не знаю, и нам, простым смертным, не нужно этого знать. Можно считать, что все — и временные, и не временные объекты — хранятся на стеке, а привязывание ссылки к rvalue делает его постоянным — *материализует*. Хотя я что-то видел про то, что в C++17 и это соптимизировали, и в некоторых случаях rvalue не материализуется, ну да ладно.
</лажа>
(Замечание от тов. Титова: понятия “prvalue”, “временный объект” и “материализация” были изменены в C++17 для введения обязательного copy elision.)

Универсальные ссылки (universal references)[52]: костыль языка, предназначенный для того, чтобы в шаблонное принимаемое значение функции можно было принимать как rvalue, так и lvalue. Выглядит это так:

```
template <typename T>
void foo(T&& t){
   /* ... */
}
```

В таком, и только таком случае, универсальная ссылка работает как универсальная ссылка. Такие записи уже работают как rvalue-ссылка:

```
template <typename T>
void foo(const T&& t){
   /* ... */
}
```

и

```
void foo(std::vector<int>&& v){
   /* ... */
}
```

Что умеет универсальная ссылка? Если в функцию передаётся lvalue:

```
template <typename T>
struct Checker{
   Checker() = delete;
```

[51] Имеется в виду “выражение, состоящее только из идентификатора”.
[52] Неофициальный термин, введённый Скоттом Мейерсом. Часто также встречается термин “forwarding references”, так как такой вид ссылок обычно используется в комбинации с std::forward.

```
};

template <typename T>
void foo(T&& t){
   Checker<T> chT;              //std::vector<int>&
   Checker<decltype(t)> cht;    //std::vector<int>&
}

int main() {
   std::vector<int> v(4);
   foo(v);
}
```

...то T выводится как type&, и по правилу сворачивания ссылок T&& → (type&) && → type&, то есть, t — lvalue-ссылка.

Если передать rvalue:

```
template <typename T>
struct Checker{
   Checker() = delete;
};

template <typename T>
void foo(T&& t){
   Checker<T> chT;              //std::vector<int>
   Checker<decltype(t)> cht;    //std::vector<int>&&
}

int main() {
   std::vector<int> v(4);
   foo(std::move(v));           //xvalue
   foo(std::vector<int>(6));    //and prvalue
}
```

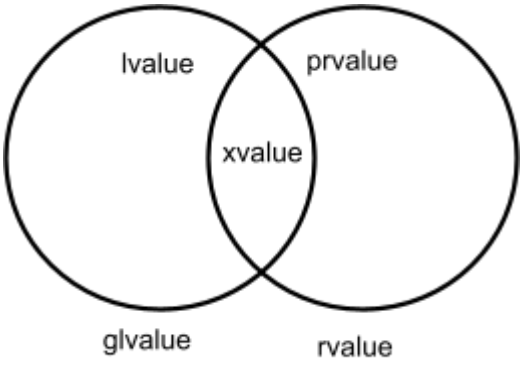
...то T выводится как type, и по правилу сворачивания ссылок T&& → (type) && → type&&, то есть, t — rvalue-ссылка.

Какие присваивания между разными видами ссылок разрешены?

```
int a = 5;                      //ok
&& к lvalue нельзя:
int &&b = a;                    //error
А к prvalue можно
int &&c = 6;                    //ok
И к xvalue
int &&d = std::move(a);         //ok
Работает как и обычная ссылка
d = 1000;                       //a = 1000 now
И если c - xvalue
int &&e = std::move(c);         //ok
e = 100000;                     //c = 100000 now
lvalue-ссылку к rvalue-ссылке - можно
int& f = c;                     //ok
Все ссылки сохраняются
f = 1;                          //e = 1 and c = 1 now
```

```
Про константность
const int&& g = 6;             //ok
Тут правила старые
int& h = g;                    //error
const int& i = g;              //ok
const int&& j = std::move(g); //ok
Очевидно
j++;                           //error
const int&& k = std::move(c); //ok
Стандартно
c = 9;                         //k = 9 now
//and so on
```

27. (review) Что такое lvalue, rvalue, xvalue, glvalue и prvalue? Приведите примеры. Расскажите о том, что такое reference collapsing (сворачивание ссылок), reference qualifiers (ссылочные квалификаторы), return value optimization (RVO) и copy elision. Приведите примеры, иллюстрирующие все вышеперечисленное.



Полное описание того, какие действия возвращают какую категорию значений, есть тут. Основные пункты:
lvalue:
1) идентификатор
2) функция, возвращающая type& (это включает перегружаемые операторы)
prvalue:
1) Конструктор (иногда)
2) функция, возвращающая type (это включает перегружаемые операторы)
xvalue:
1) результат каста к type&&
2) функция, возвращающая type&& => результат std::move и (иногда) std::forward — xvalue

Эвристика, которые многие рекомендуют: у lvalue можно взять адрес, а у xvalue и prvalue — нет

В принципе, определения lvalue и prvalue интуитивно понятны. xvalue требует дополнительных разъяснений. Можно предложить подобную практическую интерпретацию xvalue: с помощью приведения к xvalue мы помечаем lvalue как временный объект[53], что позволяет его использовать в перемещениях. Собственно расшифровка xvalue, “eXpired VALUE”, говорит сама за себя.

Выдержка из cppreference о категоризации категорий значения:

[53] см. преамбулу

With the introduction of move semantics in C++11, value categories were redefined to characterize two independent properties of expressions[5]:

- *has identity*: it's possible to determine whether the expression refers to the same entity as another expression, such as by comparing addresses of the objects or the functions they identify (obtained directly or indirectly);
- *can be moved from*: move constructor, move assignment operator, or another function overload that implements move semantics can bind to the expression.

In C++11, expressions that:

- have identity and cannot be moved from are called *lvalue* expressions;
- have identity and can be moved from are called *xvalue* expressions;
- do not have identity and can be moved from are called *prvalue* ("pure rvalue") expressions;
- do not have identity and cannot be moved from are not used[6].

The expressions that have identity are called "glvalue expressions" (glvalue stands for "generalized lvalue"). Both lvalues and xvalues are glvalue expressions.
The expressions that can be moved from are called "rvalue expressions". Both prvalues and xvalues are rvalue expressions.

Ещё несколько примеров:

```
BigInteger a(3), b(5); /* lvalue */
BigInteger c = (a + b /* prvalue */);
bar(std::move(c) /* xvalue */);
```

Reference collapsing (since C++11):
(перевод: сворачивание/сжатие ссылок)

В результате шаблонных подстановок могут возникать случаи, когда у типа больше двух амперсандов. Тогда применяются следующие правила уничтожения лишних амперсандов:

T& & → T&
T& && → T&
T&& & → T&
T&& && → T&&

где T — чистый тип без амперсандов

Если при подстановке типов получилась ссылка на ссылку, то она заменяется по следующему принципу: `T&& && → T&&`, в остальных случаях — `T&`.

Reference qualifiers:
Помните перегрузку operator+ и operator=? Которые

```
friend const BigInteger operator+(const BigInteger &first, const BigInteger &second);
BigInteger &operator=(const BigInteger &another);
```

И мы ставили const в возвращаемом значении `operator+`, чтобы нельзя было делать “(a + b) = 5;”
Но с введением move-семантики у нас может возникнуть желание написать вот так:

```
void bar(BigInteger&& a);
```

```
/* ... */
bar(a + b);
```

Что, очевидно, не скомпилируется. Поэтому было добавлено новое синтаксическое средство — reference qualifiers. После сигнатуры можно написать &, чтобы метод можно было вызывать только от lvalue, или &&, чтобы можно было вызывать только от rvalue.

В итоге получаем

```
friend BigInteger operator+(const BigInteger &first, const BigInteger
&second);
BigInteger &operator=(const BigInteger &another) &;
```

Теперь

```
void bar(BigInteger&& a);
/* ... */
bar(a + b);
```

работает, а

```
(a + b) = 5;
```

— нет, как как operator= теперь можно вызывать только от lvalue.

В итоге сигнатуры некоторых операторов должны выглядеть как-то так:

```
//Assignment operators
BigInteger &operator=(const BigInteger &another) &;
BigInteger &operator=(BigInteger &&another) & noexcept;
//Unary operators
BigInteger &operator++() &;
BigInteger operator++(int) &;
BigInteger operator+() const;
BigInteger operator-() const;
//Binary operators
BigInteger &operator+=(const BigInteger &another) &;
BigInteger &operator-=(const BigInteger &another) &;
friend BigInteger operator+(const BigInteger &first, const BigInteger
&second);
friend BigInteger operator-(const BigInteger &first, const BigInteger
&second);
```

Copy elision: принцип языка, при котором компилятор старается опускать вызов конструктора копирования или перемещения и записывать результат конструирования сразу в нужное место в памяти. Сурс

Copy elision comes in different flavors:
1) RVO — обязательна с C++17

```
struct T{
   int n;
   T(int n): n(n) {}
   T(const T& v): n(v.n){};
   T(T&& v): n(v.n){};
};

T foo(int n){
   return T(n);
}
```

```
int main() {
   // (1)
   T v1 = T(10);
   // (2)
   T v2 = foo(100);
}
```
Мещерин в первой лекции по move-семантике предлагал эти примеры как примеры, ради оптимизации которых были введены конструкторы перемещения — мол, создадутся временные объекты, которые потом переместятся. Это не так, потому что на самом деле произойдёт ещё б**о**льшая оптимизация — RVO[54]. Конструкторы вектора будут конструировать вектор сразу в той области памяти, куда этот конструктор приравнивается — сразу в v1 и v2.

Судя по cppreference, такой случай тоже вызовет лишь один конструктор:
```
T x = T(T(foo(5)));
```

3) Не RVO, но похоже — временный объект, переданный по значению
```
void bar(T x){}

int main() {
   bar(T(7));
}
```

Это можно попытаться объяснить тем, что “такая запись по сути эквивалентна
```
void bar(){
   T x = T(7);
}
```
а тут уже применима логика RVO”. Такое рассуждение *неверно*.

2) NRVO (named RVO) — когда в таком случае возвращается объект
```
T foo(int n){
   T t(n); // (1)
   /* some tweaking of t */
   return t;
}

int main() {
   T v = foo(100);
}
```
конструктор (1) сразу конструирует T на месте v из main. Обращение к t в foo — по сути, обращение к v.

4) В теории, в случае с catch тоже конструктор копирования должен опускаться
```
struct Thing{
   Thing(){}
   Thing(const Thing& another){}
};

void foo() {
   Thing c;
   throw c;
}
```

[54] до C++17 компилятор не был *обязан* делать оптимизацию, но все норм ~~пацаны~~ компиляторы её делают (GCC, Clang).

```
int main() {
    try {
        foo();
    }
    catch(Thing c) { // copy constructor is omitted
    }
}
```

Но у меня[55] тут конструктор копирования всё-таки вызывается.[56]

Пример, когда copy elision не срабатывает — множество точек выхода из функции. Не буду расписывать, так как too advanced.

28. (done) Расскажите, для решения какой проблемы нужны умные указатели. Расскажите про класс std::unique_ptr, предложите реализацию основных методов этого класса (конструкторы, деструктор, операторы присваивания, операторы унарная звездочка и стрелочка).

Все стандартные указатели определены в <memory>
Полная реализация всего, что написано далее, есть тут (правда там нет комментов, и код я ни разу не запускал, но да, я хотя бы писал аккуратно)

Проблема с обычными указателями заключается в том, что память под ними обязательно надо очищать во избежание утечки памяти. Следить за тем, чтобы каждому new соответствовал delete, муторно, часто про это забывают, а в некоторых случаях delete вызвать в принципе невозможно, например:

```
int *p = new int(5);
foo();
delete p;
```

Если `foo` бросит исключение, p никто не очистит, и произойдёт утечка памяти. Можно конечно обернуть `foo` в try-блок, но выделений памяти может быть много и в разных частях функции, перебирать все случаи — очень плохая идея. Или такой пример:

```
int *p = new int(5);
if (condition){
    delete p;
    return 4;
}
/*some code*/
if (condition){
    delete p;
    return 4;
}
/*even more code and conditions*/
delete p;
return 0;
```

Писать освобождение всей памяти в каждой ветке — замучаешься.

---

[55] Тут могут подписаться сразу несколько человек.
[56] На cppreference говорится, что это обязательная оптимизация, но выдержка из стандарта, приведённая на SO, утверждает иное.

Хотелось бы структуру, которая ведёт себя как указатель, но деструктор которой очищает память под этим указателем. Так, при выходе из области видимости указателя он будет автоматически очищаться. В C++11 с введением move-семантики реализация такой структуры стала возможна[57]

Первый, самый простой вид умного указателя - std::unique_ptr. Он реализует идею единоличного владения ресурсом под указателем. Его очень легко реализовать, но его нельзя копировать, поэтому зачастую unique_ptr не подходит

Примерная реализация[58]:

```
template<typename T>
class unique_ptr {
private:
   T *ptr;
public:
   unique_ptr(T *ptr) : ptr(ptr) {}
   //Copying is not allowed
   unique_ptr(const unique_ptr &) = delete;
   //Just take the pointer from the old object
   unique_ptr(unique_ptr &&another) noexcept {
       ptr = another.ptr;
       another.ptr = nullptr;
   };
    //Same as corresponding constructors
   unique_ptr &operator=(const unique_ptr &) = delete;
   unique_ptr &operator=(unique_ptr &&another) & noexcept{
       if (&another == this) {
           return *this;
       }
       ~unique_ptr();
       ptr = another.ptr;
       another.ptr = nullptr;
       return *this;
   };
   //Copying is not allowed, thus there will be no double delete
   T& operator*() const {
       return *ptr;
   }
   T* operator->() const noexcept {
       return ptr;
   }
   ~unique_ptr() {
       delete ptr;
   }
};
```

29. (review) Расскажите про класс std::shared_ptr. Какую проблему он решает и как он устроен? Предложите реализацию его основных методов (конструкторы, деструктор, операторы присваивания, операторы унарная звездочка и стрелочка).

shared_ptr — более умный умный указатель, который позволяет себя копировать.

---

[57] До C++11 была такая штука как std::auto_ptr, но ее нельзя было ни копировать (в привычном смысле), ни перемещать (поскольку не было move-семантики), поэтому ей мало кто пользовался

[58] Для указателей на C-style массив необходима шаблонная реализация, в которой delete[] вместо delete, нет оператора ->, но есть оператор []. Однако такая перегрузка несколько бессмысленна — есть же вектор!

Проблема: если просто копировать сырой указатель, при разрушении копий shared_ptr произойдёт delete по одному указателю несколько раз.
Идея: давайте каким-нибудь образом хранить счётчик тех shared_ptr, которые содержат в себе один и тот же сырой указатель, и сделаем так, что очищение памяти будет происходить только при разрушении последнего экземпляра shared_ptr. Это можно реализовать по-разному, но вот реализация, которая наиболее напоминает стандартную:

```
template<typename T>
   struct inner_ptr[59] {
       T *ptr;
       int count;
   };

template<typename T>
struct shared_ptr {
   //inner_ptr houses counter for how much shared_ptrs with common raw ptr
exist
   inner_ptr<T> *inner;
   //When constructing shared_ptr from raw ptr, new 'colony' of shared_ptrs
is created with new helper
   shared_ptr(T *ptr) {
       inner = new inner_ptr<T>();
       inner->count = 1;
       inner->ptr = ptr;
   }
   //Copy raw pointer, but increase shared_ptrs counter
   shared_ptr(const shared_ptr &another) {
       inner = another.helper;
       ++inner->count;
   }
   //Copy pointer, but don't alter counter because we erase pointer from
the old object
   shared_ptr(shared_ptr &&another) noexcept {
       inner = another.inner;
       another.inner = nullptr;
   }
   //Copy and swap logic: copy logic from constructor is used to avoid
copypaste, old object destructs automatically
   //It is efficient in case of complicated copy and destruction logic
   shared_ptr &operator=(const shared_ptr &another) &{
       if (&another == this) {
           return *this;
       }
       shared_ptr copy(another);
       std::swap(copy, *this);
       return *this;
   }
   shared_ptr &operator=(shared_ptr &&another) & noexcept {
       if (&another == this) {
           return *this;
       }
       inner = another.inner;
       another.inner = nullptr;
```

[59] Мещерин вспомогательный указатель называл Helper, но мне кажется, что inner_ptr — более логичное название, ведь это всё-таки внутренний указатель — тот, который прячется за инстанциями shared_ptr.

```
        return *this;
    }
    T &operator*() const noexcept {
        return *(inner->ptr);
    }
    T *operator->() const noexcept {
        return inner->ptr;
    }
    ~shared_ptr() {
        --inner->count;
        //Memory is cleaned only when there is no shared_ptrs left that hold
the same raw pointer
        if (inner->count == 0) {
            delete inner->ptr;
            delete inner;
        }
    }
};
```

Вы спр**о**сите, зачем нужен inner_ptr, ведь можно просто хранить указатель на число? Можно, но если хранить внутреннюю структуру, упрощается копирование, и один shared_ptr в итоге занимает в 2 раза меньше места.

30. (review) Предложите реализацию функции std::make_unique. Зачем нужна эта функция, какую потенциальную проблему она решает? Расскажите о функциях make_shared и allocate_shared (без реализации). Зачем нужна каждая из них, какие потенциальные проблемы они решают?

Далее пункты для unique_ptr являются подмножеством пунктов про shared_ptr, поэтому объединю их.
Конструирование типа
`shared_ptr<String> (new String("baba is you", 11));`
плохо по нескольким причинам
0) Хотелось бы вообще никогда не писать new
1) `foo(bar(), shared_ptr(new String("star vs. forces of the evil", 27)));`
Может быть так, что сперва вызывается new, потом вызывается bar, потом конструируется shared_ptr. Но bar может кинуть исключение, и произойдёт утечка памяти.[60]
2) В конструкторе shared_ptr вызывается new для inner_ptr, можно попробовать сэкономить эту аллокацию памяти

Для решения этих проблем придумали функции make_shared, make_unique (since C++14), allocate shared, allocate_unique (since C++14)

```
template <typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args){
    return unique_ptr(new T(std::forward<Args>(args)...));
}
/*somewhere is client code*/
foo(bar(), make_unique<String>("star vs. forces of the evil", 27));
auto ptr = make_unique<String>("abacaba", 7);
```

[60] Начиная с C++17, этот пункт неактуален.

Конструирование unique_ptr теперь атомарно, утечек памяти произойти не может. К тому же теперь нам вообще никогда не нужно писать new, ура!

Для shared_ptr можно сделать ещё одну оптимизацию: выделять память для inner_ptr рядом с самим объектом, а потом через placement-new конструировать inner_ptr на уже выделенном месте.

```
template <typename T, typename... Args>
shared_ptr<T> make_shared(Args&&... args){
   //allocate a big chunk of memory for object and inner_ptr together
   char* p = new char[sizeof(T) + sizeof(shared_ptr<T>::inner_ptr)];
   //Construct object
   T* ptr = new(p + sizeof(shared_ptr<T>::inner_ptr))
T(std::forward<Args>(args)...);
   //Construct tweaked shared_ptr constructor that constructs inner_ptr on
preallocated memory
   return shared_ptr(ptr, p);
}
```

shared_ptr тогда придётся немного модифицировать:

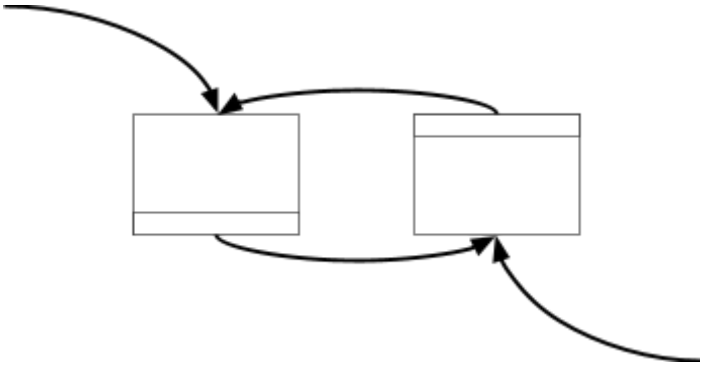
```
    //inner_ptr houses counter for how much shared_ptrs with common raw ptr
exist
template<typename T>
    struct inner_ptr {
        T *ptr;
        int count;
        //was shared_ptr constructed by make_shared or in ordinary way
        bool made = false;
    };
struct shared_ptr {
/* ... */
    //inner_ptr is constructed on already allocated place - inner_new
    shared_ptr(T *ptr, void* inner_new) {
        inner = new(inner_new) inner_ptr;
        inner->count = 1;
        inner->ptr = ptr;
        inner->made = true;
    }
/* ... */
    ~shared_ptr() {
        inner->count--;
        //Memory is cleaned only when there is no shared_ptrs left that hold
the same raw pointer
        if (inner->count == 0) {
            //If shared_ptr is made from make_shared, then we can deallocate
only one chunk of memory
            if (inner->made){
                inner->ptr->~T();
                operator delete(inner, sizeof(inner_ptr) + sizeof(T));
            } else {
                delete inner->ptr;
                delete inner;
            }
        }
    }
};
```

Ещё есть те же самые функции, только для выделения памяти на нестандартных аллокаторах. Они выглядят совершенно так же, как и из make_ аналоги, только вместо new у них std::allocator_traits<Alloc>::allocate(alloc, ...) и std::allocator_traits<Alloc>::construct(alloc, ...) или что-то такое. Соответственно, в shared_ptr появится ещё один конструктор для конструирования и удаления inner_ptr на нестандартом аллокаторе. Реализовывать их нету особого смысла, принципиально нового в них ничего нет.

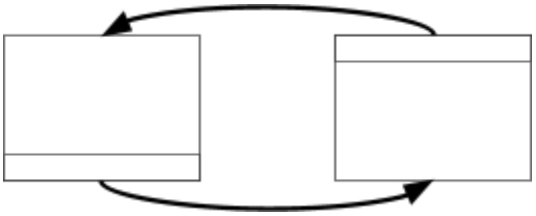
Дополнительно:
На лекциях/в билете/в программе этого нет, но я в коде часто видел метод reset(T*), который заменяет сырой указатель в shared_ptr на новый

31. (review) Расскажите про класс std::weak_ptr. Для решения какой проблемы он нужен? Предложите реализацию его основных методов. Как weak_ptr узнает, что объект под ним уже был удален?

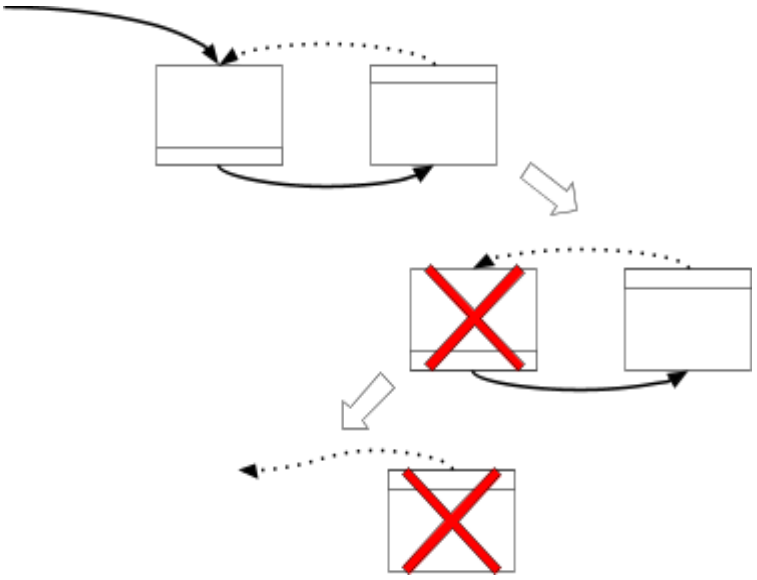
Представим такую ситуацию:



Если внешние указатели потеряются, то на каждый из двух объектов до сих пор будет указывать другой объект. Получится замкнутая система, которая никогда не удалится — утечка памяти!



Для решения этой проблемы следует заменить один из указателей на weak_ptr, который не влияет на счётчик shared_ptr в inner_ptr



Вы, конечно, разу спросите — разве в качестве указателя, который не влияет на счётчик shared_ptr, не подходит обычный указатель T*?
Ум weak_ptr заключается не в автоматическом очищении памяти как в предыдущих двух указателях, weak_ptr решает другую проблему указателей. Имея исключительно простой указатель, невозможно узнать, была ли память под ним деаллоцирована или нет. Если она уже была деаллоцирована (объект под указателем вышел из области видимости, вызов delete на копию указателя, etc), то при попытке разыменовать обычный указатель, мы получим ошибку[61]. weak_ptr решает эту проблему — он может отслеживать, был ли объект стёрт или нет. Ну, и ещё weak_ptr позволяет создавать shared_ptr, согласованный с остальными (указывающий на общий inner_ptr), чего нельзя сделать с сырым указателем.
Чтобы обратиться к объекту по weak_ptr, необходимо с помощью метода lock создать shared_ptr по тому же адресу. Если объект был удалён, то shared_ptr будет нулевым. Также состояние объекта можно проверить с помощью метода expired.

[61] Не знаю какую, скорее всего UB

Пример использования (копипаста):

```cpp
// OLD, problem with dangling pointer
// PROBLEM: ref will point to undefined data!
int* ptr = new int(10);
int* ref = ptr;
delete ptr;
// NEW
// SOLUTION: check expired() or lock() to determine if pointer is valid

std::shared_ptr<int> sptr(new int(10));
// get pointer to data without taking ownership
std::weak_ptr<int> weak1 = sptr;
// deletes managed object, acquires new pointer
sptr.reset(new int(5));
// get pointer to new data without taking ownership
std::weak_ptr<int> weak2 = sptr;
// weak1 is expired!
if(auto tmp = weak1.lock())
   std::cout << *tmp << '\n';
else
   std::cout << "weak1 is expired\n";
// weak2 points to new data (5)
if(auto tmp = weak2.lock())
   std::cout << *tmp << '\n';
else
   std::cout << "weak2 is expired\n";
```

Реализация: помимо счётчиков shared_ptr (*shared_counter*), в inner_ptr теперь будем также хранить счётчик weak_ptr (*weak_counter*). Объект будет удаляться при условии shared_counter == 0, но inner_ptr будет удаляться, только если shared_counter == 0 и weak_counter == 0; если weak_counter > 0, то какие-то weak_ptr до сих пор могут обращаться к inner_ptr и спрашивать у него, удалён ли объект или нет.
Если лочится пустой weak_ptr (который может существовать при инициализации enable_shared_from_this, смотри ниже), то кидается std::bad_weak_ptr

```cpp
template<typename T>
struct inner_ptr {
   T *ptr = nullptr; //Pointer to the object itself
   int shared_counter = 0; //how much shared_ptr is pointing to ptr
   int weak_counter = 0; //how much weak_ptr is pointing to ptr
   bool made = false;  //was shared_ptr constructed by make_shared or in
ordinary way

};

template<typename T>
struct weak_ptr {
   inner_ptr<T> *inner;
   weak_ptr() {
      inner = nullptr;
   }

   weak_ptr(const shared_ptr<T> &ptr) {
       inner = ptr.inner;
       ptr.inner->weak_counter++;
   }

   /* standard copy/move constructors/assignment operators */

   bool expired() const {
       return inner->shared_counter == 0;
   }

   shared_ptr<T> lock() const {
       if (inner == nullptr){
           throw std::bad_weak_ptr();
       }
       if (expired()) {
           return shared_ptr<T>();
       }
       return shared_ptr(inner);
   }

   ~weak_ptr() {
   /* something for case when inner is nullptr */
       inner->weak_counter--;
       if (inner->weak_counter == 0 && inner->shared_counter == 0) {
           delete inner;
       }

   }
};
```

Для работы с weak_ptr также надо добавить пару новых конструкторов в shared_ptr и дописать деструктор для обработки случаев с weak_counter

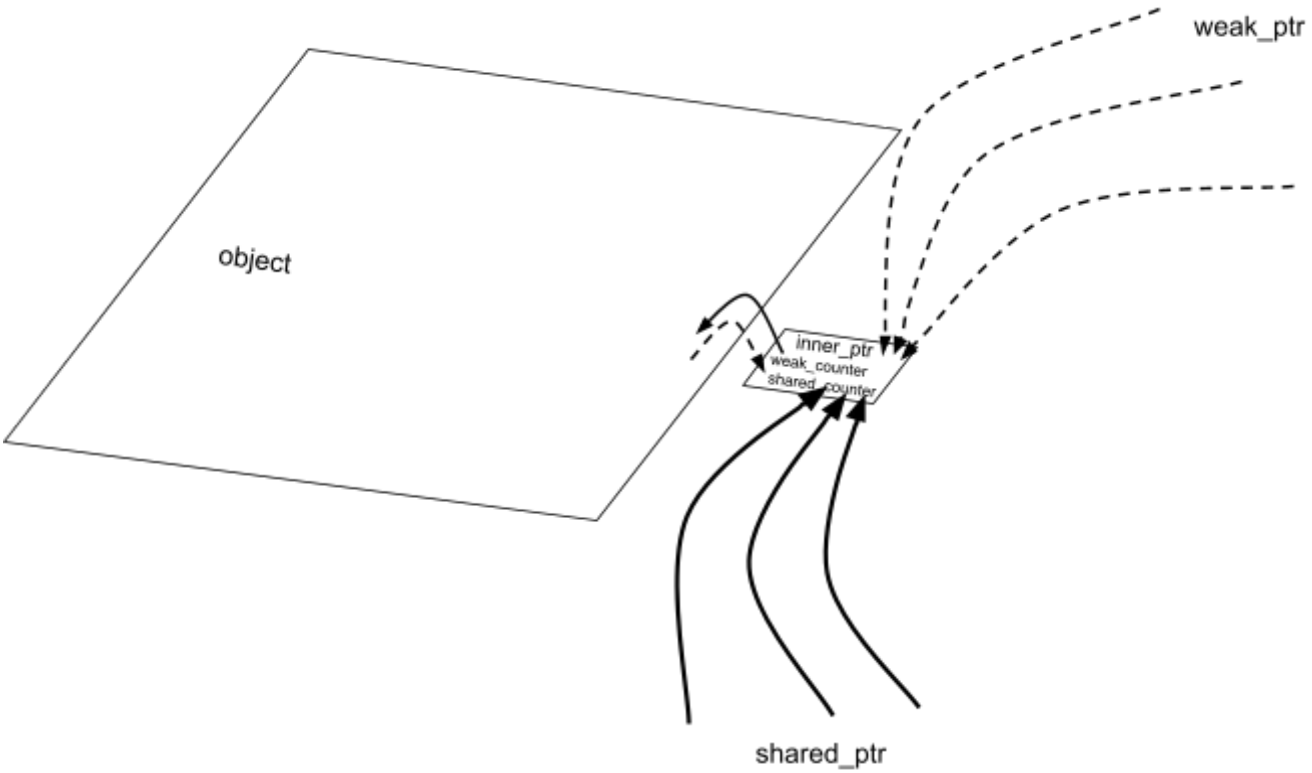
```
template<typename T>
struct shared_ptr {

   shared_ptr() = default;

   shared_ptr(inner_ptr<T> *inner) : inner(inner) {
      inner->shared_counter++;
   }

   /* ... */
   ~shared_ptr() {
       --inner->shared_counter;
       //Memory is cleaned only when there is no shared_ptrs left that hold
the same raw pointer
       if (inner->shared_counter == 0) {
           if (inner->weak_counter == 0) {
               //If shared_ptr is made from make_shared, then we can
deallocate only one chunk of memory
               if (inner->made) {
                   inner->ptr->~T();
                   operator delete(inner, sizeof(inner_ptr<T>) +
sizeof(T));
               } else {
                   delete inner->ptr;
                   delete inner;
               }
           } else {
               delete inner->ptr;
           }
       }
   }
};
```

Красивая картинка всей системы:



Дополнительно:

Зачем это нужно? Интересный ответ на stackoverflow

Почему поведение именно такое? Зачем нужны и expired, и lock? Почему lock возвращается shared_ptr, а не обычный указатель? Отличный ответ, который реально раскрывает глаза на всё (и показывает, насколько разработчики стандарта умные)

32. (review) Расскажите про класс std::enable_shared_from_this. Для решения какой проблемы он нужен и как им пользоваться? Предложите реализацию этого класса. Как нужно изменить код shared_ptr, чтобы он работал согласовано с enable_shared_from_this?

Допустим, мы хотим создать метод класса, который возвращает указатель на this. Но что делать, если мы работаем только на умных указателях и с сырыми работать не хотим? Мы же не можем каждый раз конструировать shared_ptr от this, тогда будет несколько колоний shared_ptr, из-за чего в итоге произойдёт многократное удаление. Хочется сделать такую штуку, которая будет создавать/копировать shared_ptr на this из одной колонии.

Для решения этой проблемы в стандарт добавили класс std::enable_shared_from_this, который следит за выдачей экземпляров shared_ptr. Для того, чтобы воспользоваться его функционалом, свой класс необходимо *унаследовать* от std::enable_shared_from_this, потом с помощью (публичной, кстати) функции shared_from_this можно получить shared_ptr, указывающий на this (а ещё есть функция weak_from_this!).

Куда более интересно то, как это реализовать. Идея: давайте в поле класса хранить weak_ptr на this, а в функции shared_from_this лочить его и возвращать полученный shared_ptr. Как будет инициализирован этот weak_ptr? Он будет указывать на inner_ptr, который будет сконструирован

при создании первого shared_ptr (если вызвать shared_from_this у объекта, на который не было создано shared_ptr, по стандарту [кидается std::bad_weak_ptr (since C++17)] [происходит UB (until C++17)])
Реализация enable_shared_from_this:

```
template<typename T>
struct enable_shared_from_this{
   weak_ptr<T> weak;
   /* copy/move constructors/assignment operators that do nothing */

   shared_ptr<T> shared_from_this(){
      return weak.lock();
   }

   weak_ptr<T> weak_from_this(){
      return weak;
   }
};
```

В конструкторе shared_ptr теперь нужно дописать следующее: если объект — наследник enable_shared_from_this, то инициализировать weak. Для этого придётся использовать SFINAE

```
template<typename T>
shared_ptr<T>::shared_ptr(T *ptr) {
   /* same inner_ptr init */
   //Creating weak_ptr from this shared_ptr
   if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>) {
       ptr->weak = weak_ptr(*this);
   }
}

template<typename T>
shared_ptr<T>::shared_ptr(T *ptr, void *inner_new) {
   /* same inner_ptr init */
   //Creating weak_ptr from this shared_ptr
   if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>) {
       ptr->weak = weak_ptr(*this);
   }
}
```

Дополнительно:

В билетах почему-то нет кастомного Deleter, но в программе есть, поэтому я его всё-таки распишу
Допустим вы хотите оперировать с другим ресурсом, который требует закрытия после окончания работы, например, файл или интернет-соединение. По логике напоминает unique_ptr[62], не так ли?
Поэтому разработчики языка сделали возможность подставить любую функцию вместо простого delete ptr; в деструкторе.
На самом деле в деструкторе умных указателей не вызывается явно очищение указателя, в них вызывается шаблонный функтор Deleter, по умолчанию равный std::default_delete. Реализация с учётом Deleter примерно такая:

```
template <typename T>
struct default_delete{
   void operator()(T* ptr){
```

[62] с shared_ptr — то же самое

```
        delete ptr;
    }
};

template<class T, class Deleter = default_delete<T>>
struct unique_ptr {
    T *ptr;

    /* ... */

    ~unique_ptr() {
        Deleter deleter;
        deleter(ptr);
    }
};
```

Соответственно, в кастомном Deleter можно делать что-то типа (*ptr).close()[63]
Ну и есть ещё шаблонная перегрузка std::default_delete для массивов в которой delete[] вместо delete

33. (review) Расскажите о выводе типов с помощью ключевых слов auto и decltype. В чем разница между правилами вывода типов для них? В каких контекстах можно использовать auto? Как работает decltype от идентификаторов и от разных категорий выражений (lvalue, xvalue, prvalue)? Что такое decltype(auto) и для решения какой проблемы это нужно?
1) auto — ключевое слово, которое сообщает компилятору, что тип переменной должен быть установлен, исходя из инициализируемого значения. Это работает *почти* как инстанцирование шаблонов, а именно в выражении auto x = y; слово auto следует понимать так же, как T в f(y), где f - это:

```
template<typename T>
void f(T x) {}
```

T. е. вместо T подставится тип y (в т.ч. auto&& работает так же, как и T&&, т.е. это — универсальная ссылка).

2) отличие auto от вывода типов в шаблонах
2.1) Есть такой объект std::initializer_list — инициализация с помощью фигурных скобок. Например, для вектора ввели специальный конструктор от initializer_list'а. Так вот, если написать auto x = {1, 2, 3}; , то auto станет initializer list'ом, а вот с шаблонами не так. Пример:

```
template<typename T>
void f(const T& x) { std::cout << 1; }

template<typename T>
void g(const std::initializer_list<T>& x) { std::cout << 2; }

int main() {
 f({1, 2, 3}); // compilation error: unable to make template argument for
"T"
 g({1, 2, 3}); // prints 2
}
```

2.2) auto не может использоваться в аргументах функции; auto не может использоваться в качестве возвращаемого значения функции, если в зависимости от работы функции возвращаются вещи разных типов.

---

[63] Я не знаю синтаксис, я просто предлагаю идею

3) decltype — это ключевое слово (не функция!), определяющее тип выражения/идентификатора (на этапе компиляции) и “подставляющее” его на своё место. ВАЖНО, что decltype от выражения не вычисляет его.

3.1) decltype выводит тип не так, как auto и шаблоны. decltype не отбрасывает `const`/`volatile`, *, &; НО: если под ним выражение, то есть 3 случая:

1) выражение под ним — prvalue → возвращает его тип без амперсандов;
2) lvalue → навешивает один &;
3) xvalue → навешивает &&.

3.2) Ещё важный момент: как работает decltype от тернарного оператора? Тип выражения x == 5 ? 1 : 2.0 должен быть определён ещё на этапе компиляции, вычислений не происходит, так как тогда определить тип? Так вот, есть определённые правила, по которым decltype пытается найти общий тип, к которому могут быть приведены оба варианта, а если не получается, то CE.[64]

3.3) на decltype можно навешивать модификаторы типа:

```
int main() {
 int x = 0;
 decltype(x++)& y = x;
 ++y;
 std::cout << x; // prints 1
}
```

4) decltype(auto)

Рассмотрим следующую ситуацию. Пусть есть какой-то контейнер, по которому можно перемещаться, но, возможно, это перемещение нетривиально, поэтому для этого есть отдельная функция. Но что же написать в типе возвращаемого значения? auto нельзя, ибо будет создаваться копия возвращаемого объекта (по аналогии с шаблонами); auto& тоже нельзя, ибо все, конечно, помнят про std::vector<bool>, оператор [] которого возвращает временный объект специального типа (для записи нужного бита), но тогда получится lvalue-ссылка на временный объект → словим UB. auto&& также нельзя, так как будет битая rvalue-ссылка.

Тут поможет decltype, т. е. можно написать что-то вроде decltype(c[i]) в возвращаемом значении, но до ввода переменных в область видимости к ним нельзя обращаться. Тогда в C++11 придумали синтаксис: `auto f(...) -> decltype(c[i]) {...}`, что коряво. Поэтому в C++14 ввели decltype(auto) (т. е. верни, сам догадайся что, но по правилам decltype, а не auto, т.е. не отбрасывая ссылки). Код:

```
template<typename C>
decltype(auto) getByIndex(C& c, std::size_t i) {
 std::cout << "has got the element\n";
 return c[i];
}

int main() {
 std::vector<int> v(3, 0);
 getByIndex(v, 0) = 1;  // prints: “has got the element”
 for (const auto& i : v) {
   std::cout << i << ' ';  // 1 0 0
 }
}
```

Дополнительно: ещё больше особенностей auto и decltype здесь.

Для проверки типа выражения можно использовать грязный хак[65]:

```
template<typename T>
class C {
```

[64] Это правило в некотором смысле

[65] Даже сам Скотина его использует!

```
 C();
};

int main() {
 int x = 5;
 C<decltype(++x)> c; // error: 'C::C() [with T = int&]' is private within
this context
}
```

Примечания:

1. decltype(x) выведет обычный тип x без амперсандов. (x — просто переменная: int x), потому что это идентификатор (хотя вроде как идентификатор — lvalue). А decltype((x)) выведет int&, потому что (x) — это уже выражение, возвращающее lvalue[66]

34. (review) Расскажите о возможностях compile-time вычислений в C++. Как с помощью шаблонного метапрограммирования имитировать циклы и условия? Покажите на примере проверки целого числа на простоту. Расскажите о ключевом слове constexpr: чем constexpr отличается от const, зачем нужны constexpr-функции и переменные, какими возможностями они обладают и какие ограничения на них накладываются?

1) Compile-time if, for реализуются с помощью static полей класса и специализаций шаблонов (т.к. шаблонные параметры компилятор должен разрешить пока компилирует, а static поля, инициализирующиеся константами, вычисляются на этапе компиляции). Ниже приведены 3 примера кода (C++11, заметим, что они не совсем корректные, но компилятор может на этапе компиляции проинициализировать определённое число const int переменных; В C++03 надо было бы писать без переменной p в main); условия имитируются с помощью специализаций шаблонов, циклы имитируются с помощью рекурсии в шаблонной подстановке:
имитация условия:

```
template<bool b>
void f() { std::cout << "even"; }

template<>
void f<false>() { std::cout << "odd"; }

int main() {
   const int p = 5;
   f<p % 2 == 0>(); // prints odd
}
```

проверка числа на простоту (линейный перебор):

```
template<int p, int n>
struct prime_test {
   static const bool value = (p % n != 0) && prime_test<p, n - 1>::value;
};

template<int p>
struct prime_test<p, 1> {
   static const bool value = true;
};

template<int p>
struct prime_test<p, 0> {
   static const bool value = false;
```

[66] Сос мыслом, не так ли?

```
};

template <int n>
struct is_prime {
   static const bool value = prime_test<n, n - 1>::value;
};

int main() {
   std::cout << is_prime<11>::value;
}
```

вывод последовательности от 1 до 100 (сам вывод происходит не на этапе компиляции)*:*

```
template<int n>
struct SSeq {
   static void print() {
       SSeq<n - 1>::print();
       std::cout << n << '\n';
   }
};

template<>
struct SSeq<1> {
   static void print() { std::cout << 1 << '\n'; }
};

int main() {
   const int p = 100;
   SSeq<p>::print();
}
```

2) constexpr (с C++11) — это не просто константа, а константа, значение которой должно быть известно на этапе компиляции (то есть константа времени компиляции). Вот как раз вместо `const int p = 100;` должно быть `constexpr int p = 100;` для полной корректности.
Отличия от const: это const, которая должна быть известна (вычислена) на этапе компиляции.
Возможности: Функции: C++11: одна строка, начинающаяся с return, C++14: можно делать if, for внутри constexpr функций; Условия: C++17: `if constexpr () {}`
Ограничения: нельзя проинициализировать `constexpr` через не `constexpr` — это общее ограничение. Для функций нельзя: new или delete, throw exception (в gcc баг, поэтому не работает throw, если даже на этапе компиляции оно не летит), создание нетривиальных объектов (т.е. не только простые поля, лежащие в памяти друг за другом, а, например, виртуальное наследование).
Пример кода:

```
constexpr int fib(int n) {
   if (n == 0) { return 0; }
   else if (n < 2) { return 1; }
   return fib(n - 1) + fib(n - 2);
}

int main() {
   constexpr int x = 10;
   constexpr int p = fib(x);
   if constexpr (p == 55) {
      std::cout << "yes ";
   }
   std::cout << p; //  prints yes 55
}
```

Примечания:

1. Если у компилятора что-то не получится сделать на этапе компиляции, то constexpr станут как обычные переменные/функции

35. (review) Что такое type_traits? Объясните, как реализованы структуры std::remove_reference, std::is_same, std::rank, std::conjunction, std::conditional, std::common_type.

1) type_traits — это структуры, которые помогают что-то понять про тип (с C++11). Объявлены в заголовочном файле type_traits.

2) Реализации структур:

2.1) std::remove_reference “убирает” lvalue/rvalue ссылку у типа. Код:

```
template<typename T>
struct remove_reference {
   typedef T type;
};

template<typename T>
struct remove_reference<T&> {
   typedef T type;
};

template<typename T>
struct remove_reference<T&&> {
   typedef T type;
};
```

2.2) std::is_same проверяет на равенство типы. Не работает с динамическим определением типа (т.е. когда тип переменной не совпадает с реальным типом, например, есть виртуальное наследование/функции и передача по ссылке на родителя, по которой лежит что-то другое), тип на этапе компиляции должен быть известен.

```
template <typename U, typename V>
struct is_same {
   static const bool value = false;
};

template <typename T>
struct is_same<T, T> {
   static const bool value = true;
};

int main() {
   constexpr int x = 0;
   constexpr double y = 0;
   if (!is_same<decltype(x), decltype(y)>::value) { std::cout << "same"; }
}
```

2.3) std::conjunction — это аналог логического И для типов (напомним, что есть true_type и false_type для удобства использования статических констант типа bool: достаточно просто от этих структур унаследоваться). Приведём пример кода с реализацией для переменного количества аргументов:

```
struct true_type {
   static const bool value = true;
};

struct false_type {
   static const bool value = false;
};
```

```
template <typename First, typename... Args>
struct conjunction {
   static const bool value = First::value && conjunction<Args...>::value;
};

template <typename U, typename V>
struct conjunction<U, V> {
   static const bool value = U::value && V::value;
};

struct SMyTrueType : public true_type {};

struct SMyFalseType : public false_type {};

struct SMyTrueType2 : public true_type {};

struct SMyTrueType3 : public true_type {};

int main() {
   std::cout << conjunction<SMyTrueType, SMyFalseType,
   SMyTrueType2, SMyTrueType3>::value; // prints 0
}
```

2.4) std::conditional — это аналог тернарного оператора для типов (если b == true, то он равен U, иначе равен V). Ниже реализация:

```
template <bool b, typename U, typename V>
struct conditional {
   typedef V type;
};

template <typename U, typename V>
struct conditional<true, U, V> {
   typedef U type;
};

int main() {
   constexpr int x = 6;
   conditional<x == 5, int, double>::type y = 0;
   if(std::is_same<decltype(y), int>::value){ // prints double
       std::cout << "int";
   } else {
       std::cout << "double";
   }
}
```

2.5) std::rank — возвращает число измерений массива (если не массив, то 0). Реализация (заметьте, что для массивов константного размера сделана отдельная версия, больше подробностей здесь):

```
template <typename T>
struct rank {
   static const std::size_t value = 0;
};

template <typename T>
struct rank<T[]> {
   static const std::size_t value = rank<T>::value + 1;
};
```

```
template <typename T, uint32_t N>
struct rank<T[N]> {
   static const std::size_t value = rank<T>::value + 1;
};

int main() {
   constexpr int d1 = 1, d3 = 3;
   std::cout << rank<int[d1][2][d3]>::value; // prints 3
}
```

2.6) std::common_type — определяет для всех типов общий тип, то есть такой тип, что все могут быть в него неявно преобразованы (здесь будет приведена *весьма упрощённая* реализация с лекции (лектор разрешил так отвечать) и пример использования, но правильная реализация сложнее и использует SFINAE). Реализация + пример:

```
template<typename U, typename V>
struct common_type {
 typedef decltype(true ? std::declval<U>() : std::declval<V>()) type;
};

template<class T>
struct Number { T n; };

template<class T, class U>
Number<typename common_type<T, U>::type> operator+(const Number<T>& lhs,
                                                   const Number<U>& rhs) {
 return {lhs.n + rhs.n};
}

int main() {
 std::cout << std::is_same_v<int&&, common_type<int, int>::type> << '\n';
// prints 1

 Number<int> i1 = {1}, i2 = {2};
 Number<double> d1 = {2.3}, d2 = {3.5};
 std::cout << "i1i2: " << (i1 + i2).n << "\ni1d2: " << (i1 + d2).n << '\n'
           << "d1i2: " << (d1 + i2).n << "\nd1d2: " << (d1 + d2).n << '\n';
 /* prints i1i2: 3
 i1d2: 4.5
 d1i2: 4.3
 d1d2: 5.8 */
}
```

36. (done) Что такое SFINAE? Объясните общую идею на примере структуры std::enable_if. Предложите реализацию структуры std::is_class, объясните, как она работает (проверкой на is_union можно пренебречь).

1) SFINAE — substitution failure is not an error — это правило, которым руководствуется компилятор при создании необходимого кода. Хорошая демонстрация:

```
template<typename T>
typename T::type f(T x) {
 std::cout << 1;
 return x;
}

template<typename...>
```

```
int f(...) {
 std::cout << 2;
 return 1;
}

int main() {
 f<int>(5); // prints 2
}
```

Поясним: на этапе выбора перегрузки функции происходит шаблонная подстановка. Должен был выбраться вариант с только T как более частный, но при попытке подстановки происходит ошибка (у типа int нет никакого type), поэтому выбирается вариант с произвольным числом параметров. (Ещё раз отметим, что ошибка должна возникать именно на этапе перегрузки функции, а не при выборе специализации шаблонов или в теле функции, иначе SFINAE не работает).

2) std::enable_if — шаблонная структура с двумя параметрами: bool и T. Необходима, когда нужно, чтобы тип существовал/не существовал в зависимости от условия. Реализация:

```
template<bool cond, typename T>
struct enable_if {};

template<typename T>
struct enable_if<true, T> {
 typedef T type;
};

//Пример использования немного надуманный, но всё же:
struct SA {
 static const bool val = false;
 SA(int x) {}
};

template<typename T>
typename std::enable_if<T::val, T>::type f(int x) {
 std::cout << 1;
 return 1;
}

template<typename...>
int f(...) {
 std::cout << 2;
 return 2;
}

int main() {
 f<SA>(5); // prints 2
}
```

2.1) (дополнительно) трюк с void_t нужен для включения/выключения некоторой версии шаблонного класса

```
template<class...>
using void_t = void;

template<typename U, typename V, typename = void>
struct SC {
 static const bool val = false;
```

```
};

template<typename U, typename V>
struct SC<U, V, void_t<typename U::type>> {
 static const bool val = true;
};

int main() {
 std::cout << SC<int, bool>::val; // prints 0
}
```

3) std::is_class проверяет, является ли пользовательский тип структурой или классом. Так как про union пока забыли, то считаем, что операция объявления указателя на член класса доступна только у классов и структур, поэтому достаточно попробовать объявить указатель на член.

```
class CC {
public:
 CC() : val_(0) {}

 char GetVal() { return val_; }
private:
 char val_;
};

template<typename C>
class is_class {
 template<typename...>
 static int f(...) { return 0; }

 template<typename T>
 static decltype(std::declval<int T::*     >(), bool()) f(int x) { return
1; }
public:
 static const bool value = std::is_same_v<decltype(f<C>(5)), bool>;
};

int main() {
 std::cout << is_class<CC>::value; // prints 1
}
```

Дополнительно: полезным окажется следующий пример (он наглядно показывает практическое применение SFINAE).
Для проверки наличия метода у класса можно создать структуру, принимающую в шаблонах класс и список аргументов метода, внутри которой будет перегруженная функция (более общая версия — 1 и более частная — 2). В версии 2, чтобы она выбиралась первой, попробовать запустить то, что не скомпилируется при отсутствии метода, а в версии 1, например, ничего не делать. Тогда из-за того, что при перегрузке функций не рассматривается тип возвращаемого значения, можно возвращать в разных версиях функции разные типы. Соответственно типом возвращаемого значения и будет определяться значение value.
Тем, что не будет компилироваться при отсутствии метода, будет примерно следующее выражение:

```
decltype(std::declval< T>().construct(std::declval< Args>()...), int())
```

Объясним его смысл. Тут написан тип возвращаемого значения версии f: оператор запятая от двух выражений → нужно упростить сначала левое выражение, потом правое. declval (см. след. вопрос) не реализована, ну и пускай, она возвращает тип C&&. У типа C&& вызвали бы construct с

такими-то Args. А какой тип возвращает construct? Если construct не реализован, то ошибка и используется SFINAE, а если реализован, то посмотрим, что идёт вторым: ага, там int. Ок, тип всего выражения — int.
Тогда весь код будет выглядеть как-то так:

```
// Check for the existence of a method with necessary arguments.
struct SC {
   void construct(int n, char c) {
       std::cout << "I'm constructor, " << n << c;
   }
};

template <typename T>
T&& declval();

// Value is true <=> in class C there is a method taking Args.
template <class C, typename... Args>
class has_method_construct {
private:
   template <typename...>
   static char f(...) { return 1; }
   // Do smth, that can be compilated in one cases and can't in others.
   // P.S. we have to imitate a call of the method.
   template <typename T, typename... Kwargs>
   static decltype(declval<T>().construct(declval<Kwargs>()...), int())
f(int x) { return 1; }
public:
   static const bool value = std::is_same_v<decltype(f<C, Args...>(5)),
int>;
};

int main() {
   std::cout << has_method_construct<SC, int>::value; // prints 0
}
```

37. (review) Зачем нужна функция std::declval? Почему эта функция возвращает T&&, а не T? Предложите реализацию структуры std::is_constructible, объясните, как она работает.
1.1) std::declval
Идея: написать функцию, которая будет “возвращать объект” типа T&&. Нужна для корректного вывода типов на этапе компиляции. На самом деле, у неё нет реализации, так как declval используется в контексте, где выражение не вычисляется (она не подразумевает использования в контексте реального выполнения — суть её в том, чтобы от объекта данного типа писать какие-то выражения и проверять, компилируются они или нет).
1.2) declval возвращает T&& из-за неполных типов (очень условно: нельзя создавать элементы этого типа, потому что, например, нет тела класса, или деструктор удалён). Есть правило, по которому даже в контексте, не подвергающемуся вычислению, иногда нельзя использовать выражения, имеющие неполный тип (например, в функциональном вызове), но T&& уже не неполный тип. Также заметим, что T&& “не портит” тип ссылки, а вот T& может свернуть ссылку (навешивание на T&& & даст T&). В разделе “Дополнительно” в предыдущем вопросе есть пример самописной declval, а вот пример, который падает с ошибкой “*использование неопределенного типа CC*”:

```
class CC;
```

```
bool f(CC&& x) { return false; }

template<typename T>
T declval() {}

int main() {
 std::cout << std::is_same_v<decltype(f(declval<CC>())), int>;
}
```

2) std::is_constructible — проверяет, правда ли тип C имеет конструктор с данными аргументами (с точностью до приведения типов). Реализуется почти точно также, как и дополнительный пример из предыдущего номера, с той лишь разницей, что
`decltype(declval<_T>().construct(declval<_Args>()...), int()) f(int x) { return 1; }`
заменится на: `decltype(_T(declval<_Args>()...), int()) f(int x) { return 1; }`
Все пояснения по работе даны в том же примере.

Дополнительно: реализация std::is_nothrow_move_constructible — проверка noexcept конструирования C от каких-то rvalue.
Тут нам потребуется проверка move конструктора, но помимо этого ещё нужно уметь проверять на на noexcept. Вспомним, что на этот счёт есть std::conditional, а также оператор noexcept, который проверяет, бросает ли потенциально хоть какое-нибудь исключение выражение (также можно написать и без std::conditional, см. ниже).

```
struct SC {
  int val;
  explicit SC(SC&& other) noexcept {
    val = other.val;
  }
};

template<class C>
struct is_nothrow_move_constructible {
private:
  template<typename...>
  static int f(...) { return 0; }

  template<typename T>
  static typename
  std::conditional<noexcept(T(std::declval<T>())), bool, int>::type f(int
x) {
    return true;
  }
public:
  static const bool value = std::is_same_v<decltype(f<C>(5)), bool>;
};

int main() {
  std::cout << is_nothrow_move_constructible<SC>::value; // prints 1
}
```

Без *std::conditional* private будет выглядеть так:

```
template<class C>
struct is_nothrow_move_constructible {
private:
  template<typename...>
  static constexpr int f(...) { return 0; }
```

```
  template<typename T>
  static constexpr decltype(T(std::declval<T>()), int()) f(int x) {
    return noexcept(T(std::declval<T>())) ? 1 : 0;
  }

public:
  static const bool value = f<C>(5) == 1;
};
```

38. (review) Зачем нужен класс std::allocator_traits? Предложите реализацию функции allocator_traits<Allocator>::construct. Каким образом эта функция проверяет, реализован ли у аллокатора метод construct, и как она себя ведет в зависимости от этого?

1) std::allocator_traits нужен для стандартизированного доступа к методам аллокаторов: std::allocator_traits<Alloc>::allocate (а также deallocate, construct, destroy). Один из посылов к созданию этого класса — одинаковость методов construct и destroy для всех аллокаторов (стандартный вариант — вызов placement-new от указателя с нужными параметрами для конструктора и вызов деструктора по указателю).

2) std::allocator_traits<Alloc>::construct проверяет наличие метода construct у пользовательского аллокатора с помощью SFINAE. Если метод отсутствует, то вызывает placement-new. *Настоятельно рекомендуется ознакомиться с примером из "Дополнительно" к 36-ому вопросу и повторить тему "Аллокаторы"*, так как там дано подробное описание всего, что тут будет использоваться. Здесь же ограничимся примером кода реализации и использования (только для метода *construct()*) с краткими комментариями.

Пример кода:

```
// Allocator, that can construct T from char.
template <typename T>
struct SMyAllocator {
    T* allocate(std::size_t n) const {
        return static_cast<T*>(::operator new(sizeof(T) * n));
    }
    void construct(T* ptr, char& c) {
        std::cout << "I'm char constructor: "  << c << '\n';
        ::new(ptr) T(std::forward<char>(c));
    }
    void deallocate(T* ptr, std::size_t n) const {
        ::operator delete(ptr, n);
    }
    void destroy(T* ptr) const {
        ptr->~T();
    }
};

// a static constant of bool type
template<bool _Val>
using bool_constant = std::integral_constant<bool, _Val>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;

template <class Alloc>
struct SAllocator_traits {
    template <class T, typename... Args>
```

```
    static void construct1(false_type x, Alloc& a, T* ptr, Args&&... args)
{
        ::new(ptr) T(std::forward<Args>(args)...);
        std::cout << "operator new was called\n";
    }
    template <class T, typename... Args>
    static void construct1(true_type x, Alloc& a, T* ptr, Args&&... args) {
        a.construct(ptr, std::forward<Args>(args)...);
    }

    template <class T, typename... Args>
    static void construct(Alloc& a, T* ptr, Args&&... args) {
        construct1(bool_constant<has_method_construct<Alloc, T*,
Args&&...>::value>(), a, ptr, std::forward<Args>(args)...);
    }
};

int main() {
    SMyAllocator<int> alloc;
    int* x = alloc.allocate(1);
    int* y = alloc.allocate(1);
    char sym = 'a';
    bool b = false;

    SAllocator_traits<SMyAllocator<int>>::construct(alloc, x, sym); //
prints: I'm char constructor: a
    SAllocator_traits<SMyAllocator<int>>::construct(alloc, y, b); //
prints: operator new was called

    alloc.destroy(x), alloc.deallocate(x, 1);
    alloc.destroy(y), alloc.deallocate(y, 1);
}
```

Комментарии: в данном коде используется реализация *has_method_construct* из примера в *дополнительно* к 36-му вопросу. По поводу std::integral_constant могу напомнить, что *operator()* возвращает *value*, а также там определён C-style cast к шаблонному типу, который возвращает *value*.
Стоить объяснить, как устроен *SAllocator_traits*. Стандартный метод *construct* запускает *construct1* в зависимости от того, что лежит в *has_method_construct*. Если там *value == true*, то нужно запускать пользовательский *construct*, иначе — *placement_new*.

39. (done) Расскажите о функции std::move_if_noexcept. В каком месте стандартной библиотеки она используется и почему она там необходима? Предложите ее реализацию (считая, что type_traits уже реализованы).
1) std::move_if_noexcept по сути “определяет”, когда безопасно вызывать перемещающий конструктор, а когда нужно копирующий.
2) Пример из библиотеки: как известно, std-шные СД должны давать гарантию безопасности при исключениях. Для строгой гарантии как раз и используется эта функция в реализации std::push_back в векторе (проблема: делается move эл-тов вектора, вдруг полетело исключение; гарантируется по стандарту, что данные останутся в прежнем состоянии, но часть массива уже move’нулась, что делать -_-).
3) Код + пример (если что, за забытие std::is_copy_constructible карать не будут, так сказал Мещерин):

```
template<typename T>
```

```
typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
std::is_copy_constructible<T>::value,
                          const T&,
                          T&&>::type move_if_noexcept(T& x) {
 return static_cast<typename std::conditional<
     !std::is_nothrow_move_constructible<T>::value &&
std::is_copy_constructible<T>::value, const T&, T&&>::type>(x);
}

struct Bad {
 Bad() = default;
 Bad(Bad&&) { // may throw
   std::cout << "Throwing move constructor called\n";
 }
 Bad(const Bad&) { // may throw as well
   std::cout << "Throwing copy constructor called\n";
 }
};

struct Good {
 Good() = default;
 Good(Good&&) noexcept { // will NOT throw
   std::cout << "Non-throwing move constructor called\n";
 }
 Good(const Good&) noexcept { // will NOT throw
   std::cout << "Non-throwing copy constructor called\n";
 }
};

int main() {
 Good g;
 Bad b;
 Good g2 = move_if_noexcept<Good>(g); // Non-throwing move constructor
called
 Bad b2 = move_if_noexcept<Bad>(b); // Throwing copy constructor called
}
```

Есть смысл разъяснить, почему именно такие ссылки используются в таких-то местах.
В возвращаемом типе:

1) const T& — чтобы было точное соответствие типов для copy конструктора (скорее всего, T& тоже можно, но есть вероятность, что что-то пойдёт не так);
2) T&& — чтобы было соответствие типов с move-конструктором.

В принимаемом типе:

- T, const T — бессмысленно, ибо произойдёт создание нового объекта;
- const T&& — бессмысленно, ибо есть const T&;
- const T& — нельзя, ибо кастовать к T&& нельзя будет;
- T&& — нельзя, ибо тогда это универсальная ссылка, для которой другие правила вывода типов (на выходе там, где T&&, получим T&).

40. (review) Предложите реализацию структуры std::is_base_of. Объясните, как она работает.

1) std::is_base_of проверяет, является ли данный тип частным случаем второго типа (т.е. наследником)

Идея: если Base — родитель Derived, то можно скастовать указатель на Derived к указателю на Base. Код:

```
class CBase {};
class CDerived : private CBase {
  int val;
};

template <class Base, class Derived>
struct SHelper {
  operator Derived*();
  operator Base*() const;
};

template <class Base, class Derived>
struct is_base_of {
  template <class T>
  static int f(Derived*, T);

  static char f(Base*, int);

  static const bool value = std::is_same_v<decltype(f(SHelper<Base,
Derived>(), int())), int>;
};

int main() {
  std::cout << is_base_of<CBase, CDerived>::value;
}
```

2) Как работает: у компилятора есть 3 варианта запуска f:

1) скастовать Helper к Derived* (касты он как бы умеет), подставить вместо T int, тем самым сделав 1 каст, 1 подстановку;

2) скастовать Helper к Derived*, Derived* скастовать к Base* (если можно), выбрать f(Base*, int), сделав 2 каста и не делая шаблонную подстановку;

3) скастовать Helper к const Helper, const Helper к Base* константным образом и вызвать f(Base*, int), сделав 2 каста, один из которых const, и не делая шаблонную подстановку.

Случаи:

1) Derived — наследник Base. Тогда для второй версии функции есть 2 варианта приведения аргументов. При выборе из 2) и 3) предпочтительнее не const cast, поэтому выбирается 2). А уже из 1) и 2) при перегрузке выбирается 1) (так следует из правил стандарта).

2) Derived — НЕ наследник Base. Тогда нет 2) => только 3). Но тут работает уже правило, что более частный случай лучше, поэтому вариант с шаблонами не выбирается.

41. (done) Что такое лямбда-функции в C++? Каков синтаксис лямбда-выражений? Что такое списки захвата, что такое захват по умолчанию и в чем его опасность? Расскажите о дополнительных возможностях лямбда-выражений в C++14 (захват с инициализацией, обобщенные лямбда-выражения). Расскажите (без реализации) про класс std::function и функцию std::bind, приведите примеры их использования.

(Новичкам рекомендуется к прочтению статья для понимания происходящего, ниже изложен материал для повторения, дополненный деталями из лекции)

1) Лямбда-функциями называются безымянные локальные функторы, которые можно создавать прямо внутри какого-либо выражения.

Синтаксис:

```
[список захвата] (аргументы) /*опционально: mutable*/
/*опционально: noexcept*/ /*опционально: "-> тип возвращаемого значения"*/
{ тело функции };
```

Чаще всего лямбды используются в качестве функторов и являются удобным сокращением последних (хотя если функтор имеет сложную и длинную реализацию, то всё же лучше вынести её в отдельный класс). Например, следующая строка возвращает количество четных элементов вектора:

```
std::count_if(vec.begin(), vec.end(), [] (int n) { return (n % 2) == 0; });
```

2) Списки захвата

2.1) По умолчанию объекты, доступные в текущей области видимости, не видны в теле лямбды (вспомним, что лямбды не что иное как `operator()` внутри анонимного функтора), поэтому для использования в теле лямбды их надо указывать в списке захвата (`[]`). Захватывать можно только объекты, доступные в текущей локальной области видимости, с *ограниченным* временем жизни (automatic storage duration). В частности это значит, что глобальные и локальные статические (см. билет 3) объекты не могут быть захвачены. Это обусловлено тем, что к таким объектам в теле лямбды можно обратиться в любое время и указывать их в списках захвата просто не имеет смысла[67]. Пример:

```
int GLOBAL = 5;
int main() {
 static int stat = 0;
 [] (int a) {
   stat = 10;
   GLOBAL = 1;
 };
}
```

Существует захват по значению:

```
[a, b] () {};
```

И по ссылке:

```
[&a, &b] () {};
```

По умолчанию захваченные *по значению* переменные являются `const`. Если необходимо их изменять, пишется квалификатор `mutable`:

```
[a] () mutable { ++a; };
```

Захваченные *по ссылке* переменные можно менять по умолчанию.

2.2) Захватом по умолчанию называется следующая конструкция:

```
[=] () {}; //захват по умолчанию по значению
[&] () {}; //захват по умолчанию по ссылке
```

Такие лямбды захватывают *все* доступные для захвата объекты. Захват по умолчанию является плохим стилем и не рекомендуется к использованию. Вот почему:

Во-первых, захват по умолчанию по ссылке может привести к проблеме битых ссылок[68]. От этой проблемы, вообще говоря, никто не застрахован и при явном перечислении переменных в списке захвата, но так как при таком захвате объектов в глобальной области видимости много, забыть о продолжительности жизни используемого объекта гораздо проще.

Во-вторых, есть тонкий момент с работой лямбд в классе:

---

[67] Это можно объяснить тем, что анонимный функтор генерируется "в месте объявления" лямбды.

[68] Т.к. время жизни лямбды может превысить время жизни локального объекта. Например, если лямбда была объявлена статическим полем глобального класса и захватывает по & некоторые локальные переменные.

★ *При захвате по умолчанию лямбдой в теле класса захватывается указатель на this.*
То есть поле класса в теле лямбды на самом деле разворачиваются в this->"поле". Это приводит к тому, что от проблемы битых ссылок нельзя избавиться даже при захвате по умолчанию по значению. Так, код:

```
class C {
 public:
  auto GetFoo() {
    return [=](int n) { return n % divisor; };
  }
 private:
  int divisor = 5; //приватные поля также доступны в лямбде
};

int main() {
  auto f = C().GetFoo();
  std::cout << f(7);
}
```

Также приведёт к UB, ибо при вызове `f(5)` произойдёт подстановка `divisor` → `this->divisor`, где `this` уже разрушен.

3) Возможности с C++14.
3.1) Захват с инициализацией похож[69] на списки инициализации в конструкторах: он позволяет инициализировать захватываемые переменные выражениями. В частности, это позволяет захватывать переменные по &&:

```
int a = 0;
[a = std::move(a)]() { auto b = a; };
```

3.2) Обобщённые лямбда выражения — "шаблонные" лямбды. В отличие от обычных функций, в лямбдах можно использовать auto в *аргументах*:

```
[](auto a, auto&& b) {}; //причем b является универсальной ссылкой!
```

Допустим, нужно, чтобы process:

```
[](auto &&x) { process( /*как передать x?*/ ) };
```

правильно обработал x в зависимости от того, lvalue это или rvalue. Разбором случаев можно убедиться, что данная конструкция даст нужное количество &:

```
[](auto &&x) { process(std::forward<decltype(x)>(x)); };
```

4) std::function и std::bind (объявлены в `<functional>`)
4.1) std::function — шаблонный класс, являющийся "обёрткой" над функциями. Экземпляры std::function могут хранить, копировать, и ссылаться на любой вызываемый объект (функцию).
Синтаксис:

```
std::function</*тип возвращаемого значения*/(/*аргументы*/)> foo = ...
```

Например:

```
bool foo(int a, int b) { return a == b; }
int main() {
 std::function<bool(int, int)> f = foo;
}
```

4.2) std::bind позволяет создавать проекцию функции многих переменных. Например, если есть функция 2-х аргументов, то эта функция при фиксированном первом аргументе является уже функцией одной переменной.
Синтаксис:

---

[69] Уточним: не просто похож, а *является* списками инициализации в конструкторах, ибо список захвата — конструктор анонимного функтора.

```
void f(int x, char y, double z) {
 std::cout << x << ' ' << y << ' ' << z << '\n';
}

int main() {
 //  g - проекция f на 2 и 3 координаты, т.е. функция одной переменной
 std::function<void(int)> g = std::bind(f, std::placeholders::_1, 'a', 10);
 //  h меняет местами 1 и 3 аргументы[70]
 std::function<void(double, char, int)> h = std::bind(f,
std::placeholders::_3, std::placeholders::_2, std::placeholders::_1);
 g(1);  // выведет 1 a 10
 h(1, 'b', 2.1);  // выведет 2 b 1
}
```

Дополнения:

1. Если лямбда функция имеет сложную систему return-ов (к примеру, возвращает выражения разных типов (например, int и float)), то после аргументов ставится `->` и возвращаемое значение.
   ```
   [] (int a) -> double {
    if (a < 5)
      return a + 1.0;
    else if (a % 2 == 0)
      return a / 2.0;
    else
      return a * a;
   };
   ```
2. Подробнее о захвате полей классов. Поля классов нельзя захватывать напрямую (можно захватить только this), т.е. следующая конструкция есть CE (сам класс см. в 2.2)):
   ```
   auto GetFoo() {
    return [divisor](int n) { return n % divisor; };
   }
   ```
   Тем не менее следующая конструкция работает (и UB уже не будет, т.к. divisor уже локальная переменная лямбды, скопированная с this->divisor):
   ```
   auto GetFoo() {
    return [divisor = divisor](int n) { return n % divisor; }; //первый
   divisor - захваченная по значению переменная, второй - поле класса
   }
   ```
3. Тип *каждой* лямбды определяется компилятором самостоятельно, а это значит, что единственным способом создавать объекты *типа* лямбд вляется `auto`:
   ```
   auto f = [](int a) { std::cout << "a: " << a; };
   ```
   Можно написать и так:
   ```
   std::function<void(int)> f = [](int a) { std::cout << "a: " << a; };
   ```
   но это не значит, что тип лямбды есть `std::function`!
4. Примеры реального использования лямбд (функции определены в <algorithm>):
   1) Возвращает количество четных элементов вектора.
   ```
   std::count_if(vec.begin(), vec.end(), [] (int n) { return (n % 2) ==
   0; });
   ```
   2) Умножает элементы вектора на 2.
   ```
   std::for_each(vec.begin(), vec.end(), [] (int& n) { n *= 2; });
   ```
5. `std::is_invocable<Function, Args…>` — проверяет, правда ли function можно вызвать

[70] Placeholder'ы — специальная структура, имитирующая аргументы.

с такими аргументами.
`std::is_function<Class>` — проверяет, являются ли объекты класса callable.

42. (review) Что такое юнионы (union), для чего они нужны? В чем сходство и различие между юнионами и классами? Каковы особенности инициализации полей юниона, что такое “активный член” юниона и как его поменять? Что такое std::variant и std::any, для чего они нужны, каковы их основные методы?

1) union
1.1) нужен для экономии памяти. Внешне union-ы достаточно похожи на структуры/классы (далее будут примеры), но есть большое различие в работе union: оно не создаёт для каждого поля место в памяти, а создаёт место, достаточное для размещения самого большого объекта, и использует только эту память. Для лучшего понимания рекомендую посмотреть видео (всего 12 минут), здесь же кратко ответим на поставленные вопросы.
1.2) сходства: есть поля, есть функции-члены, на которые можно создавать указатели (!, тут мы вспоминаем про std::is_class и нашу реализацию, к которой нужно добавить проверку на union), да и вообще тут проще сказать, что схоже всё то, что не различно.
1.3) различия:
3.1) В структуре элементы располагаются в памяти последовательно друг за другом, а объединение позволяет хранить различные типы данных в одном и том же пространстве памяти (но не одновременно);
3.2) union не может содержать виртуальные методы;
3.3) union не может наследоваться или быть наследуемым;
3.4) union не должен иметь статических членов данных или членов данных ссылочного типа;
3.5) If any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), move constructor (12.8), copy assignment operator (12.8), move assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-provided or it will be implicitly deleted (8.4.3) for the union.
1.4) активный член — тот, с которым ведётся работа на данный момент (в частности, было присваивание значения этому члену), в других полях будет лежать что-то (см. видео), но вообще обращение к неактивным членам — это UB.
Пример кода:

```
struct SA {
    int a;
    double b;

    SA(int x) : a(x), b(0.0) {}
    SA(double x) : b(x), a(0) {}
    SA() : a(0), b(0.0) {}
};

union SUnion {
    int x;
    SA y;
    int* p;

    SUnion() {}
    SUnion(int other) : y(other) {}
    SUnion(int*& other) : p(other) { other = nullptr; }
    ~SUnion() {}
};
```

```
int main() {
   int* x = new int();
   *x = 15;
   SUnion u2 = x;
   *u2.p = 10;
   std::cout << u2.x << ' ' << *u2.p << ' '; // prints 15294832 10
   // do not forget about memory leak before changing the active member
   delete u2.p;
   u2.y = 40000;
   std::cout << u2.x << ' ' << u2.y.a << ' ' << u2.y.b; // prints 40000
40000 0
}
```

Пояснения: по первому выводу: так как указатель имеет какое-то значение, то вот x его и взял; по второму: тут хорошо видно, как распределена общая память: активный член — структура SA, в которой сначала лежит int a == 40000, потом double b = 0.0, так как x неактивный, в нём значение из первых 4-ых байт, то 40000.

2) std::variant (C++17) — по сути то же, что и union, только безопаснее и удобнее (variant может быть одним из типов, перечисленных в шаблоне). Теперь не нужно каждый раз вызывать конструкторы через placement-new и деструкторы, как было с union.
Можно добавить следующее: “Variant is not allowed to allocate additional (dynamic) memory.

A variant is not permitted to hold references, arrays, or the type void. Empty variants are also ill-formed (std::variant<std::monostate> can be used instead).

A variant is permitted to hold the same type more than once, and to hold differently cv-qualified versions of the same type.

Consistent with the behavior of unions during aggregate initialization, a default-constructed variant holds a value of its first alternative, unless that alternative is not default-constructible (in which case the variant is not default-constructible either).”

Основные методы:

2.1) все операторы сравнения (сравнивают variant-ы как то, что под ними лежит);

2.2) std::get (шаблонная функция, читает значение variant-а с учётом индекса или типа (если уникален), иначе кидает ошибку);

2.3) std::holds_alternative (шаблонная функция, проверяет, содержит ли variant заданный тип). Ещё больше всего тут.

3) std::any (C++17) — это объект произвольного, неизвестного типа (там может лежать всё, что угодно).

Основные методы:

3.1) type() — возвращает typeid хранящегося значения;

3.2) std::any_cast — шаблонная функция, которая бросает исключение, если тип, переданный в шаблоне, не является типом хранящегося в any объекта;

3.3) has_value() — проверяет на наличие содержимого any.

Дополнительно:

1. Скажем пару слов о возможной реализации std::variant. Сначала нужно понять, сколько памяти нужно выделить (сделать поле типа void* максимальным размером; для нахождения размера нужно, как, например, в printf постепенно разворачивать шаблоны и делать от каждого sizeof). Потом придумать, как при присвоениях менять тип (сначала через вспомогательную функцию проверить, существует ли тип, совпадающий с типом присваиваемого значения, вернуть его; далее проверить, нужно ли вызывать нетривиальный деструктор у текущего значения (std::is_trivially_destructible); а потом менять тип поля через reinterpret_cast, делать присвоение).
2. Скажем пару слов о возможной реализации std::any. Что туда положить и какого размера? Понятное дело, что нужно размещать всё в динамической памяти, ибо объект может быть любого размера. Тогда нужно хранить указатель нужного типа, чтобы реализовывать удаление по нему при присваивании нового значения, но any не шаблонный => нужна доп структура Helper<T>, которая будет хранить указатель и удалять по нему. Но опять же, так как any не шаблонный, мы не можем хранить этот указатель внутри него. Поэтому возможный вариант — наследовать Helper от BaseHalper с виртуальным деструктором и хранить внутри any указатель на BaseHelper. Кандидат на код есть в продвинутом вопросе.
3. Также стоит упомянуть std::optional (C++17), который либо хранит значение типа T, либо ничего не хранит. Умеет проверять, есть ли под ним значение, возвращать значение и ещё кое-чего. Удобен в функциях, которые могут ничего не возвращать, или для полей класса, которые в зависимости от условий могут не существовать. Пример кода:

```
std::string good = "Success";
class nullopt {};

template <typename T>
class optional {
private:
 bool initialized = false;
 T value;
public:
 optional(const T& other) : value(other), initialized(true) {}
 optional(const nullopt& other) {}
 bool has_value() const { return initialized; }
 T& get_value() {
   if (initialized) { return value; }
   else throw 1;
 }
};

optional<std::string> create_string(bool b) {
 if (b) { return good; }
 else { return nullopt(); }
}

int main() {
 try {
   std::cout << create_string(false).get_value(); // prints: “optional
was empty”
 }
 catch (int x) {
   std::cout << "optional was empty";
 }
}
```

4. std::reference_wrapper — это объект, который ведёт себя как ссылка *идеологически*, но имеет полноценные конструктор, деструктор, оператор присваивания и т.д.
(например, можно создать вектор из reference_wrapper, т.е. вектор из как-бы-ссылок и работать с ним как с вектором ссылок, а вот вектор из обычных ссылок создать нельзя, ибо как минимум тип int&* некорректен).
Пример кода, возможно, будет позднее, но в программе этот пункт выделен красным.

43. (review) Что такое рефлексия? Что можно сказать о возможностях рефлексии в C++? Реализуйте шаблонную функцию detect_fields_count<S>, которая бы позволяла определять количество полей у данной структуры S. Объясните, как это работает.

1) Рефлексия — это, образно говоря, взгляд на что-то изнутри наружу (например, есть рефлексия в психологии, когда человек анализирует свои ошибки, т.е. смотри на себя со стороны).
В программировании — это возможность узнавать что-то про конкретный объект, модифицировать код программы во время её работы из самой программы
(примеры: поиск и модификация конструкций исходного кода (блоков, классов, методов, интерфейсов (протоколов) и т. п.) как объектов первого класса во время выполнения; изменение имён классов и функций во время выполнения; анализ и выполнение строк кода, поступающих извне)

2) В C++ нет полноценной рефлексии, т.е. нельзя, например, в коде программы узнать, как называются поля какого-то класса, какие поля приватные, а какие публичные, сколько аргументов у такой-то функции и т.п. Но кое-что сделать можно! (См. следующий пункт).

3) detect_fields_count<S>
Предположение: нет никакого наследования, структура не содержит никаких виртуальных функций и т.п. (в общем, Plain Old Data). Чтобы Вы не думали, что это бесполезная функция, скажем, что изначальный мотив — написать вывод в поток для любой POD структуры (поэтому нужно знать число полей, их типы и значения, а потом по типу и номеру поля в структуре делать вывод).
Идея: давайте попробуем проинициализировать структуру очень большим количеством каких-то вещей; пользуясь SFINAE, можно пытаться инициализировать до тех пор, пока не получится (причём нужно подбирать не только количество, но и правильный тип). Пример кода (разбор далее):

```
struct SB {
    int x;
    double y;
    std::string str;
    bool b;
};

struct SUniversalType {
    template <typename T>
    constexpr operator T();
};

template <std::size_t I>
using SUniversalType_constructor = SUniversalType;

// SUniversalType_constructor<I0>{} - SUniversalType creation without
parameters
template <typename S, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t &out,
std::index_sequence<I0, I...>) noexcept
->decltype(S{ SUniversalType_constructor<I0>{},
SUniversalType_constructor<I>{}... }) {
    out = sizeof...(I) + 1;
    return S{};
```

```
}

template <class S, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out,
std::index_sequence<I...>) noexcept {
   detect_fields_count<S>(out, std::make_index_sequence<sizeof...(I) -
1>{});
}

int main() {
   SB s = { 1, 2.0, "abc", true };
   constexpr int start_fields_count = 10;
   std::size_t ans = 0;

   detect_fields_count<SB>(ans,
std::make_index_sequence<start_fields_count>{});
   std::cout << ans; // prints 4
}
```

Создадим вспомогательную структуру SUniversalType, в которой будет как бы определён оператор приведения к любому типу T. Тогда большим количеством объектов этой структуры будем пытаться проинициализировать структуру S. Используем SFINAE: в менее общей версии в возвращаемом значении будем пытаться инициализировать поля структуры через {universal_type$_0$, ..., universal_type$_{N-1}$}. Так как universal_type кастится к любому типу, проблема только в количестве. Если получилось, сохраняем число шаблонных параметров (т.е. число “инициализаторов” => ответ). При ошибке будет вызываться более общая версия, в которой вызывается менее общая с числом шаблонных параметров на один меньшим.
Для того, чтобы контролировать текущее число “инициализаторов”, будем работать с ними через шаблонные параметры и количество передаваемых в функцию аргументов. Для этого достаточно обычной последовательности натуральных чисел в передаваемых значениях и использования variadic templates с non-type template parameters, так что в этом нам поможет std::integer_sequence (*std::index_sequence* — последовательность чисел времени компиляции (*constexpr* чисел)). И да, заметьте, что SUniversalType_constructor<I>{} превратится в создание пустой структуры типа SUniversalType благодаря *using*.

Список ошибок в видеолекциях

От Мещерина

За 26 лекций лектором (т.е. мной), к сожалению, было допущено существенное количество ошибок - как мелких неточностей, так иногда и довольно концептуальных. Многие из ошибок исправлялись в ходе лекций и эти исправления попали на видеозапись, а другие - нет. Этот список нужен для того, чтобы дополнить видеолекции и исправить детали, которые на видеозаписи были проговорены неверно.

В этом списке ошибки перечислены не в хронологическом порядке, а в порядке убывания критичности (по субъективному мнению лектора).
2. В лекции 11 про новые спецификации исключений было сказано, что noexcept-функция, которая кидает исключенибудет terminate.
3. В лекции 9 про шаблоны в примере специализации шаблона спутана специализация с перегрузкой, пример надо исправить
4. В лекции про dynamic_cast и typeid не было сказано, что типы должны быть полиморфными для применения к ним typeid
6. Неточность в лекции 18, когда обсуждалась реализация make_shared: на самом деле make_shared не вызывает new напрямую, а тоже делает это через аллокатор (а именно, вызывает allocate_shared со стандартным аллокатором).
7. Неточность в лекции 19: про enable_shared_from_this было сказано, что если shared_ptr на данный объект еще не существует, то shared_from_this может его создать. На самом деле нет, эта функция требует, чтобы хоть один shared_ptr от объекта уже существовал к моменту ее вызова.
[TO BE CONTINUED]

От нас

Далее идёт список неточностей, замеченных авторами этого конспекта (как подтверждённые, так и потенциальные):
1) 9, remove_const: не надо у неспециализации ставить <T>
2) 10, Variadic templates: забыто многоточие в сигнатуре функции f(t...)
3) 10, CRTP: CLion не догадался и забыта ссылка, нужно было прописать Base<Derived> *b = &d;
4) 9, template with using: не надо у alias писать шаблон, т.е. просто
using mytype = …
5) Лекция 13. Не существует std::allocator_traits<Alloc>::select_on_container_move_construction, -//-_copy_assignment и -//-_move_assignment. Есть только select_on_container_copy_construction, propagate_on_container_copy_assignment и propagate_on_container_move_assignment (причем вторые работают не совсем не так (см в реализации вектора 11 пункт)
6) Вероятно ошибка. Лекция 15 std::distance вызывается с шаблонными параметрами в <> (а не в аргументах!), хотя частичная специализация для функций запрещена.
7) Лекция 8. Нельзя писать `virtual void s() = 0 {}`
Если хочется написать тело чисто виртуальной функции, то (не) делайте это вне класса

Правила оформления документа

- Код вставляем:
  - `courier new`
  - 11 масштаб
  - одинарный межстрочный интервал
  - внешний вид кода: darcula из CLion (Если нет возможности такого оформления, вставлять нейтральный текст)
  - заливка: весь фрагмент цветом фона текста (т.е. darcula) (формат/стили абзацев/границы и заливка/цвет фона/”в нижней таблице 1-й сверху (а лучше

посмотреть какой цвет в других местах стоит, ибо он меняет позицию время от времени)”).

- Относящиеся к теме, но не к билету подробности указываются в таком стиле:
  Дополнения:
  1. …
  2. ...

  и т.д.

- Незаконченный вопрос (билет) помечается `(wip)`. Один билет пишет один человек (или, по крайней мере, человек, пишущий билет, должен разрешить совместное редактирование). Законченный билет помечается `(review)`. Далее для того, чтобы билет перешёл в стадию `(done)` хотя бы 2 человека должны проверить билет. Проверяющий может редактировать билет с *сохранением прошлого варианта* (новый вариант заливается ярким цветом для наглядности). В отредактированных местах оставляется комментарий с замечаниями. Билет считается проверенным, если проверяющий не имеет замечаний (в таком случае он ставит пометку в виде комментария “Проверил”).

- Если код превышает по размерам страницу A4, то вставляем ссылку на Gist.

На этом моё знание плюсов заканчивается