3.1-3.2 Понятие компилятора. Примеры компиляторов C++. Понятие ошибок компиляции (CE) с примерами. Понятия ошибок времени выполнения (RE) и неопределенного поведения (UB) с примерами. Почему не любую RE компилятор может предвидеть? Почему не любое UB приводит к RE?

Понятие компилятора

Компилятор – это программа, которая переводит текст, написанный на языке программирования, в набор машинных кодов.

- Компилируемые языки:
  * Программа на компилируемом языке при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в *исполняемый файл*, который может быть запущен на выполнение как отдельная программа. Другими словами, компилятор переводит программу с языка высокого уровня на низкоуровневый язык, понятный процессору сразу и целиком, создавая при этом отдельную программу

  Примерами компилируемых языков являются Pascal, C, C++, Rust, Go.

- Интерпретируемые языки
  * Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ее текст без предварительного перевода. При этом программа остается на исходном языке и не может быть запущена без интерпретатора. Можно сказать, что процессор компьютера — это интерпретатор машинного кода. Кратко говоря, интерпретатор переводит на машинный язык прямо во время исполнения программы.

  Примерами интерпретируемых языков являются PHP, Perl, Ruby, Python, JavaScript. К интерпретируемым языкам также можно отнести все скриптовые языки.

Note: Java и C, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код.

Примеры компиляторов C++:

1. GNU Compiler Collection aka GCC
2. Clang
3. C++ Builder
4. Microsoft Visual C++

Запуск компилятора из командной строки: **g++ main.cpp** или **clang++ main.cpp**
Запуск исполняемого файла из командной строки: **./a.out**

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. **Лексические:** ошибка в процессе разбиения на токены, т.е. компилятор увидел последовательность символов, которую не смог расшифровать.

   ✓ `(std) (::) (cout) (<<) (x) (;)` – пример корректного разбития на токены

   × `24abracadabra;`

2. **Синтаксическая:** возникает, когда вы пишете инструкцию, недопустимую в соответствии с грамматикой языка (например служит речь Мастера Йоды)

   × `std::<int>vector x;`

   × `x + 5 +;`

   × нет точки с запятой (;)

   × несоответствие круглых или фигурных скобок

3. **Семантическая:** возникате, когда инструкция написана корректно, но компилятор ее выполнить не может (например: съешьте себя этим столом)

   × использование необъявленных переменных

   × вызов метода `size()` от переменной типа int

   × `x++ = a + b;`

   × вызов `foo(3);` хотя сигнатура такая: `void foo(int a, int b);`

Runtime error aka RE

Программа компилируется корректно, но в ходе выполнения она делает что-то непотребное. RE невозможно отследить на этапе компиляции (компилятор может разве что кинуть предупреждение в месте потенциальной ошибки). Примеры:

× Слишком большая глубина рекурсии – **stack overflow ⇒ segmentation fault**

× Слишком далекий выход за границу массива – **segmentation fault**

× Целочисленное деление на ноль (не всегда компилятор такое может предвидеть)

× Исключение, которое никто не поймал – RE

Замечание: не всякое исключение есть RE, и не каждое RE есть исключение.

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:

× Повторное удаление одного объекта.

× Битые ссылки — ситуация, когда используется ссылка на разрушенный (чаще всего из-за выхода из области видимости) объект. Использование такой ссылки является UB.

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

int main() {
    int a = foo();
    // can be anything, but more likely this will cause seg fault
}
```

× `int x = 2 << 40;` – не определено, что будет происходить при переполнении знакового типа.

× Чтение выделенной, но неинициализированной памяти. В теории, считается какой-то мусор, но технически, так как это UB, компилятор в праве поджечь ваш монитор.

× Отсутствие **return** в конце функции, которая что-то возвращает. Шок, да? Это UB!

× Недалекий выход за границы **C-style массива**.

Замечание: К сожалению, математически нельзя сделать все ошибки CE. За счёт UB в C++ мы выигрываем в эффективности.

3.3-3.4 Основные типы. Приоритет операций.

C++ является статически-типизированным языком, то есть на момент компиляции все типы должны быть известны. Основные типы, с которыми мы сталкивались:

Тип	*байт*	*Диапазон значений*
логический тип данных		
bool	1	$[0; 2^8)$
целочисленные типы данных		
short	2	$[-2^{15}; 2^{15})$
unsigned short	2	$[0, 2^{16})$
int	4	$[-2^{31}; 2^{31})$
unsigned int	4	$[0; 2^{32})$
long long	8	$[-2^{63}; 2^{63})$
unsigned long long	8	$[0; 2^{64})$
вещественные типы данных		
float	4	$[-2^{31}; 2^{31})$
double	8	$[-2^{63}; 2^{63})$
long double	10	.....
символьные типы данных		
char	1	$[-2^7; 2^7)$
unsighned char	1	$[0; 2^8)$

Так же используются **литеральные суффиксы:**

- **.u** для unsigned int
- **.ll** и **.ull** для long long и unsigned long long
- **.f** для float
- и прочее

Integer promotion: "меньший" тип приводится к "большему"
Замечание: Неявное преобразование к unsign: может вызывать проблемы, например:

```
int x = -1;
unsigned y = 0;
std::cout << x + y;  // very big number
```

Note: size_t беззнаковый целый тип данных, возвращаемый оператором sizeof, определен в заголовочном файле `<cstring>`
Note: Сложение char - это UB

Выражения и операторы

В следующей таблице перечислены приоритет и ассоциативность операций C++. Операции перечислены сверху вниз в порядке убывания приоритета.

Приоритет	Оператор	Описание	Ассоциативность
1	`::`	Разрешение области видимости	слева направо
2	`a++ a--`	Суффиксный/постфиксный инкремент и декремент (возвращает копию старого объекта)	
	тип`()` *тип*`{}`	Функциональный оператор приведения типов	
	`a()`	Вызов функции	
	`a[]`	Индексация	
	`. ->`	Доступ к полю или методу класса (через объект класса / указатель на объект)	
3	`++a --a`	Префиксный инкремент и декремент (возвращает ссылку на уже измененный объект)	Справа налево
	`+a -a`	Унарные плюс и минус	
	`! ~`	Логическое НЕ и побитовое НЕ	
	(*тип*)	Приведение типов в стиле C	
	`*a`	Косвенное обращение (разыменование)	
	`&a`	Взятие адреса	
	`sizeof`	Размер в байтах[примечание 1]	
	`co_await`	Выражение await (C++20)	
	`new new[]`	Динамическое распределение памяти	
	`delete` `delete[]`	Динамическое освобождение памяти	
4	`.* ->*`	Указатель на элемент	Слева направо
5	`a*b a/b` `a%b`	Умножение, деление и остаток от деления	
6	`a+b a-b`	Сложение и вычитание	
7	`<< >>`	Побитовый сдвиг влево и сдвиг вправо	
8	`<=>`	Оператор трёхстороннего сравнения (начиная с C++20)	
9	`< <=`	Операторы сравнения < и ≤ соответственно	
	`> >=`	Операторы сравнения > и ≥ соответственно	
10	`== !=`	Операторы равенства = и ≠ соответственно	
11	`&`	Побитовое И	
12	`^`	Побитовый XOR (исключающее или)	
13	`|`	Побитовое ИЛИ (включающее или)	
14	`&&`	Логическое И	
15	`||`	Логическое ИЛИ	
16	`a?b:c`	Тернарный условный оператор[примечание 2]	Справа налево
	`throw`	Оператор throw	
	`co_yield`	Выражение yield (C++20)	
	`=`	Прямое присваивание (предоставляется по умолчанию для классов C++)	
	`+= -=`	Составное присваивание с сложением и вычитанием	
	`*= /= %=`	Составное присваивание с умножением, делением и остатком от деления	
	`<<= >>=`	Составное присваивание с побитовым сдвигом влево и сдвигом вправо	
	`&= ^= |=`	Составное присваивание с побитовым И, XOR и ИЛИ	
17	`,`	Запятая (возвращает то, что справа)	Слева направо

При синтаксическом анализе выражения оператор, указанный в некоторой строке приведённой выше таблицы, будет более тесно связан со своими аргументами (как в случае применения скобок), чем любой оператор из строк, расположенных ниже, с более низким приоритетом. Например, выражения `std::cout<<a&b` и `*p++` будут разобраны как `(std::cout<<a)&b` и `*(p++)`, а не как `std::cout<<(a&b)` и `(*p)++`.

Операторы с одинаковым приоритетом связываются со своими аргументами в направлении их ассоциативности. Например, выражение `a = b = c` будет разобрано как `a = (b = c)`, а не `(a = b) = c`, так как ассоциативность присваивания справа налево, но `a + b - c` будет разобрано как `(a + b) - c`, а не `a + (b - c)`, так как ассоциативность сложения и вычитания слева направо.

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)

Литералы — последовательность символов, интерпретируемая как константное значение какого-то типа (1, ‘a’, “abc”, 0.5, true, nullptr, etc)

Операторы — это, можно сказать, функции со специальными именами (`=, +, <, [], ()`)

Выражение — некоторая синтаксически верная комбинация литералов и идентификаторов, соединенных операторами

Использования оператора запятая

```
    int x = 0;
    int y = 2;
    int z = (++x, ++y);
    std::cout << z;  // it will be 3

    // BUT!
    int sum = add(x, y); // this is not a comma operator
    int x=3, y=5; // this one also
```

Использования тернарного оператора

Тернарный оператор выделяется из ряда других операторов в C++. Его называют *”conditional expression”*. Ну а так как это *expression*, выражение, то как у каждого выражения, у него должен быть **тип** и **value category**.

Типом тернарного оператора будет **наиболее общий тип** его двух последних операндов. (Например, у int и short общим типом будет int.) Т.е. наиболее общий тип это такой тип, к которому могу быть приведены оба операнда. Вполне могут быть ситуации, когда общего типа нет.

```
    struct C{};
    struct D{};
    (true ? C() : D());  // this will cause CE
```

Так. С типом тернарного оператора мы немного разобрались. Осталось решить вопрос с value category. Тут действует следующее **правило:** если в тернарном операторе происходит преобразование типов к наиболее общему, то тернарный оператор — *rvalue*, иначе *lvalue*.

```
    int i;
    int j;
    (true ? i: j) = 45;  // OK

    short i;
    int j;
    (true ? i: j) = 45; // CE
    // Здесь происходит преобразование типов.
    // Значит value category у выражения слева от знака -"- rvalue.
    // A rvalue, как известно, нельзя присваивать.


    int a = 1;
    int b = 1;
    int c = 1;
    a = true ? ++b : ++c;
    // a = 2, b = 2, c = 1
```

Где нельзя использовать if{...} else{...}, но можно тернарный оператор?

Например, в списке инициализации конструктора и при инициализации ссылки в зависимости от условия. (Как известно, нельзя объявлять не инициализированную ссылку)

```
    // 1-ый пример
    struct S {
        int i_;

        // вот так НЕЛЬЗЯ, это CE
        S() : if(some_condition) i_(1) else i_(0) {}

        // а вот так можно :)
        S() : i_(some_condition ? 1 : 0) {}
    };



    // 2-ой пример
    int a = 3, b = 4;
    int& i;

    // вот так НЕЛЬЗЯ, это CE
    if(some_condition) i = a;
    else i = b;

    // а вот так можно :)
    int& i = (some_condition ? a : b);
```

3.5 Синтаксис использования конструкций if. . . else, switch, for, while, do... while. Использование команд break, continue, return

Использование конструкции if... else:

```
if (bool condition) {
    statement1;
}
else {
    statement2;
}
```

Использование конструкции switch:

```
switch(expr) {
    case 1:
        statement1;
        break;
    case 2:
        statement2;
        break;
    ...
    default:
        defstat;
        break;
}
```

Использование конструкции for:

```
for (decl or expr; bool-expr; expr) {
        statements;
        continue;
    }
```

Использование конструкции while:

```
    while (condition) {
        expr;
    }
```

Использование конструкции do... while:

```
do {
    expr;
} while (condition);
```

Команда break используется для выхода из цикла, continue — для перехода на следующую итерацию цикла, return — для выхода из функции.

3.6 Разница между объявлением и определением. Синтаксис объявления и определения переменных, функций, классов и структур, алиасов для типов (с помощью слова using). Правило одного определения (ODR) для функций и для переменных, объяснение его смысла.

Declaration (Объявление). При объявлении чего-то всё, что вы делаете, это говорите компилятору, что есть что-то с определенным именем и определенного типа.)

Definition (Определение). Определение чего-то означает предоставление всей необходимой информации для создания этого целиком.)

Таким образом, разница в том, что после определения чего-то объект обязательно жизнеспособен. Можно объявлять, но не определять - например функции:

```
int f(); //declaration but not definition
int g(){ //definition
    f();
}
int f(){ //definition
    g();
}
```

Что можно объявлять?

1. Переменные
2. Функции
3. Классы
4. Структуры
5. Union
6. Namespace
7. Alias (Другое название для чего-нибудь, т.е. псевдоним (см. пример с using)

Синтаксис объявления и определения:
type id [=init]; - объявление переменной [определение].
type func_id($type_1 id_1, ..., type_n id_n$); - объявление функции (добавляем фигурные скобки для определения). Пример объявления:

```
namespace N {}
typedef vvi vector<vector<int>>;
using vvi =  vector<vector<int>>;
```

One Definition Rule. Оно гласит много чего, в частности функции и переменные должны быть определены ровно один раз. Формально - Only one definition of any variable, function, class type, enumeration type, concept (since C++20) or template is allowed in any one translation unit (some of these may have multiple declarations, but only one definition is allowed).)

3.7 Разница между стековой (автоматической) и динамической памятью. Динамическое выделение памяти с помощью оператора new (в стандартной форме). Освобождение динамической памяти. Проблема утечек памяти. Проблема двойного удаления.

Автоматическая память (стек)

Непосредственное управление автоматической памятью — выделение памяти под локальные объекты при их создании и освобождение занимаемой объектами памяти при их разрушении — осуществляется компилятором. Собственно, по этой причине память и называют автоматической.

Период времени от момента создания объекта до момента его разрушения, т. е. период времени, в течение которого под объект выделена память, называют временем жизни объекта.

Время жизни локальных объектов определяется областью видимости их имён. Область видимости локального имени начинается с места его объявления и заканчивается в конце блока, в котором это имя объявлено. Область видимости аргументов функции ограничивается телом функции, т. е. считается, что имена аргументов объявлены в самом внешнем блоке функции.

Статическая память

Когда программа запускается, ОС выделяет под неё память. В этой памяти есть три раздела (и не только): data, text (секция кода), stack (4 мБ). В data хранятся статические переменные - их время жизни определено сроком жизни программы.

Динамическая память

Допустим, мы хотим выделить в runtime дополнительную память. Для этой цели подходит оператор new: type* ptr = new type(size). Такое выделение памяти будет существовать до тех пор, пока мы явно не укажем, что хотим его очистить, для этого существует оператор delete. Эти операторы – достаточно низкоуровневый инструмент.

Связанные проблемы: утечки памяти (утерян указатель на выделенную память или память не была очищена), очищение памяти, которая не была выделена (когда по ошибке очистка памяти делается по иной области памяти), проблема двойного удаления (в программе по одному и тому же указателю дважды очищается память)

3.8. Понятие области видимости и времени жизни объекта. Конфликты имен переменных, замещение менее локального имени более локальным именем. Пример ситуации неоднозначности при обращении к переменной.

Непосредственное управление автоматической памятью – выделение памяти под локальные объекты при их создании и освобождение занимаемой объектами памяти при их разрушении – осуществляется компилятором. Собственно, по этой причине память и называют автоматической.
Период времени от момента создания объекта до момента его разрушения, т. е. период времени, в течение которого под объект выделена память, называют временем жизни объекта.
Время жизни локальных объектов определяется областью видимости их имён. Область видимости локального имени начинается с места его объявления и заканчивается в конце блока, в котором это имя объявлено. Область видимости аргументов функции ограничивается телом функции, т. е. считается, что имена аргументов объявлены в самом внешнем блоке функции.

Примеры:

1. Глобальная область
2. Область пространства имен
3. Локальная область
4. Область класса

```
int main() {
    int a;

    {
        int b;
        // здесь доступны переменные a и b
    }

    {
        int a;
        // более локальная переменная перебивает менее локальную
    }
}
```

Пример неоднозначности при обращении к переменной:

```
#include <iostream>

namespace A {
  int x = 1;
}

namespace B {
  int x = 2;
}

using namespace A;
using namespace B;

```

```cpp

int main() {
  std::cout << x; // CE (error: reference to 'x' is ambiguous)
}
```

3.9. Указатели и допустимые операции над ними. Сходства и различия между указателями и массивами.

Указатель — переменная, значением которой является адрес ячейки памяти. Шаблон: type* p. Требует 8 байт для хранения (чаще всего).

Операции, которые поддерживает указатель:

1. Унарная звёздочка, разыменование: $T* -> T(*p)$
   Возвращает значение объекта

2. Унарный амперсанд: $T- > T*(\&p)$
   Возвращает адрес объекта в памяти

3. +=, ++, −−, -=

4. ptr + int

5. ptr - ptr, который возвращает разницу между указателями (ptrdiff_t)

Еще есть указатель на void, void* обозначает указатель на память, под которым лежит неизвестно что. Его нельзя разыменовать.

```cpp

#include <iostream>

struct Foo {
    void bar() {
        std::cout << "bar";
    }
}

int main() {
    int* a = new int();
    int* b = new int();
    Foo* foo = new Foo();

    // Операции:
    std::cout << *a; // разыменование
    foo->bar(); // вызов метода или поля у сложного типа
    std::cout << *(a + 1); // прибавление числа
    std::cout << b - a; // разница между указателями ($ptrfdiff_t$)

    void* x = nullptr; // значение по умолчанию

    int* array = new int[10];
    std::cout << array[3] == *(array + 3); // true

    // Освобождение памяти - различие у указателя и массива:
    delete a;
    delete[] array;

}
```

Грань между массивами и указателями весьма тонкая. Массивы являются собственным типом вида int(*)[size]. Что можно делать с массивом:

1. Привести массив к указателю (это будет указатель на первый элемент)
2. К массиву можно обращаться по квадратным скобкам, как и к указателю. a[0] == *(a+0).

Для удаления массива необходимо использовать delete[]. Формально это другой оператор.

3.10. Ссылки. Объяснение концепции. Отличия от указателей. Особенности инициализации ссылок, присваивания ссылкам. Передача аргументов по ссылке и по значению. Проблема висячих ссылок, пример ее возникновения.

Мотивация: если объявление нового объекта это новое имя для старого, то компилятору необходимо было бы решать, когда удалять старый объект и хранить в runtime дополнительную информацию.

Можно считать, что ссылка - это просто переименование объекта, и код никак не различает ссылку и сам объект. (На самом деле есть способ это сделать, но не очень-то и нужно).

В отличие от указателя, ссылка не может быть пустой, она всегда должна на что-то ссылаться.
Отличия указателя от ссылки:

- Нельзя объявить массив ссылок. (any kind of arrays)
- У ссылки нет адреса. (no references to references)
- no pointers to references. Примеры:

```
    // this WILL NOT compile
    int a = 0;
    int&* b = a;

    // but this WILL
    int a = 0;
    int& b = a;
    int* pb = &b; //pointer to a

    // and this WILL
    int* a = new int;
    int*& b = a; //reference to pointer - change b changes a

```

- Существует арифметика указателей, но нет арифметики ссылок.
- Ссылка не может быть изменена после инициализации.

- Указатель может иметь «невалидное» значение с которым его можно сравнить перед использованием. Если вызывающая сторона не может не передать ссылку, то указатель может иметь специальное значение nullptr (т.е. ссылка, в отличии от указателя, не может быть неинициализированной):

```
    void f(int* num, int& num2) {
        if(num != nullptr) {} // if nullptr ignored algorithm
        // can't check num2 on need to use or not
    }

```

- Ссылка не обладает квалификатором const

```
    const int v = 10;
    //int const r = v; // WRONG!
    const int& r = v;

    enum {
        is_const = std::is_const<decltype(r)>::value
    };

    if(!is_const) \\ code will print this
        std::cout << "const int& r is not const\n";

```

Проблема висячих ссылок

```
#include <iostream>

int& bad(int x) {
  ++x;
  return x;
}


int main() {
  int y = bad(0); // время жизни объекта закончилось, а ссылки - нет
  std::cout << y; // UB
}
```

3.11. Константы, константные указатели и указатели на константу. Константные и неконстантные операции. Константные ссылки, их особенности, отличия от обычных ссылок.

Определение. *Константы* - такой тип, к которому применимы константные операции.

```
    const int x = 3;
```

Причем константы необходимо инициализировать сразу при объявлении.
Возможна передача неконстантной версии, куда необходима константая.

Определение. *Константый указатель* - это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется ключевое слово const между звёздочкой и именем указателя:

```
    int value = 11;
    int* const ptr = &value;
```

Подобно обычным константным переменным, константный указатель должен быть инициализирован значением при объявлении. Это означает, что он всегда будет указывать на один и тот же адрес. В вышеприведенном примере *ptr* всегда будет указывать на адрес value (до тех пор, пока указатель не выйдет из области видимости и не уничтожится):

```
    int val1 = 14;
    int val2 = 88;
    int* const ptr = &val1;//ок, инициализация адресом val 1
    ptr = &val2;// - не ок, после инициализации нельзя менять константый указатель
```

Однако, поскольку переменная *val*, на которую указывает указатель, не является константой, то её значение можно изменить путем разыменования константного указателя:

```
    int val = 11;
    int* const ptr = &val;
    *ptr = 8;//ок, так как тип данных (int) - не константый
```

Определение. *Указатель на константное значение* — это неконстантный указатель, который указывает на неизменное значение. Для объявления указателя на константное значение, используется *ключевое слово const перед типом данных:*

```
    const int val = 11;
    const int* ptr = &value;//ок, ptr - неконстантый указатель на const int
    *ptr = 8;//нельзя, т.к. мы не можем изменить константое значение
```

Можно также инициализировать неконстантым значением:

```
    int val = 11;
    const int* ptr = &value;
```

Указатель на константную переменную может указывать и на неконстантную переменную (как в случае с переменной *val* в примере, приведенном выше). Подумайте об этом так: указатель на константную переменную обрабатывает переменную как константу при получении доступа к ней независимо от того, была ли эта переменная изначально определена как *const* или нет. Таким образом, следующее в порядке вещей:

```
    int val = 11;
    const int* ptr = &val;//ptr указывает на const int
    val = 8;//все ок, val - не константа
```

Но не следующее:

```
    int val = 11;
    const int *ptr = &val; // ptr указывает на "const int"
    *ptr = 12; // ptr обрабатывает value как константу, поэтому изменение значения
    //переменной val через ptr не допускается
```

Указателю на константное значение, который сам при этом не является константным (он просто указывает на константное значение), можно присвоить и другое значение:

```
    int val1 = 11;
    const int *ptr = &val1; // ptr указывает на const int
    int val2 = 12;
    ptr = &val2; // хорошо, ptr теперь указывает на другой const int
```

Наконец, можно объявить константный указатель на константное значение, используя *ключевое слово const как перед типом данных, так и перед именем указателя:*

```
    int val = 11;
    const int *const ptr = &val;
```

Константный указатель на константное значение нельзя перенаправить указывать на другое значение также, как и значение, на которое он указывает, — нельзя изменить.

Определение. *Константное выражение* - это то, которое можно посчитать в compile time.

Более подробно о полной классификации можно прочитать здесь.

Определение. *Константная ссылка.* Объявить ссылку на константное значение можно путем добавления ключевого слова const перед типом данных:

```
    const int value = 7;
    const int &ref = value; // ref - это ссылка на константную переменную value
```

Ссылки на константные значения часто называют просто *«ссылки на константы»* или *«константные ссылки»*.

В отличие от ссылок на неконстантные значения, которые могут быть инициализированы только неконстантными l-values, ссылки на константные значения *могут быть инициализированы неконстантными l-values, константными l-values и r-values*:

```
    int a = 7;
    const int &ref1 = a; // ок: a - это неконстантное l-value
    const int b = 9;
    const int &ref2 = b; // ок: b - это константное l-value
    const int &ref3 = 5; // ок: 5 - это r-value
```

Как и в случае с указателями, константные ссылки также могут ссылаться и на неконстантные переменные. При доступе к значению через константную ссылку, это значение автоматически считается const, даже если исходная переменная таковой не является:

```
    int value = 7;
    const int &ref = value; // создаем константную ссылку на переменную value
    value = 8; // ок: value - это не константа
    ref = 9; // нельзя: ref - это константа
```

Когда константная ссылка инициализируется значением r-value, время жизни r-value продлевается в соответствии со временем жизни ссылки:

```
   int somefcn()
    {
        const int &ref = 3 + 4;
        /*обычно результат 3 + 4 имеет область видимости выражения
        и уничтожился бы в конце этого стейтмента, но, поскольку
        результат выражения сейчас привязан к ссылке на константное значение,*/
        std::cout << ref; // мы можем использовать его здесь
    } // и время жизни r-value продлевается до этой точки,
    //когда константная ссылка уничтожается
```

Ссылки на константные значения особенно полезны в качестве параметров функции из-за их универсальности. Константная ссылка в качестве параметра позволяет передавать неконстантный аргумент l-value, константный аргумент l-value, литерал или результат выражения:

```
    #include <iostream>
    void printIt(const int &a)
    {
        std::cout << a;
    }

    int main()
    {
        int x = 3;
        printIt(x); // неконстантное l-value
```

```
        const int y = 4;
        printIt(y); // константное l-value
        printIt(5); // литерал в качестве r-value
        printIt(3+y); // выражение в качестве r-value
        return 0;
}
```

3.12. Неявное приведение типов. Явное приведение типов с помощью оператора static_cast.

Определение. *Неявное преобразование типов*, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.

Неявное преобразование типов (или «автоматическое преобразование типов») выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию (не использует явное преобразование типов через операторы явного преобразования).

```
int foo(int x) {
    return x + 1;
}

int main() {
    double d = 3.0;
    foo(d); // здесь произошло неявное приведение типов
    int y = static_cast<int>(d); // явно попросили компилятор привести типы
}
```

static_cast - создание новой сущности из старой. Работает на этапе компиляции. Берёт объект старого типа и возвращает нового.

static_cast и неявное преобразование работает также и с пользовательскими типами, нужно только определить правила преобразования типов.

Билет 3.13. Понятие перегрузки функций. Что является и что не является частью сигнатуры функции.Основные правила выбора наиболее подходящей функции: частное лучше общего, точное соответствие лучше приведения типа. Пример неоднозначности при обращении к функции.

Def. Перегрузка функций – возможность создавать несколько одноименных функций, но с различными параметрами.

Def. Сигнатура функции – это часть общего объявления функции, которая позволяет идентифицировать эту функцию среди других. Функция распознаётся компилятором по последовательности типов её аргументов и её имени, что и составляет сигнатуру или сигнату функции. И если функция — это метод некоторого класса, то в Signature участвует и имя класса.

В сигнатуре учитываются только типы параметров, а не типы возвращаемых значений. Перегруженные функции не могут различаться лишь типами возвращаемого значения; если списки параметров двух функций разнятся только подразумеваемыми по умолчанию значениями аргументов, то второе объявление считается повторным.

```
// объявления одной и той же функции
int max ( int *ia, int sz );
int max ( int *ia, int = 10 );
```

Ключевое слово typedef создает альтернативное имя для существующего типа данных, новый тип при этом не создается. Поэтому если списки параметров двух функций различаются только тем, что в одном используется typedef, а в другом тип, для которого typedef служит псевдонимом, такие списки считаются одинаковыми.

Следующие два объявления также считаются одинаковыми:

```
// объявляют одну и ту же функцию
void f( int );
void f( const int );
```

Спецификатор const важен только внутри определения функции: он показывает, что в теле функции запрещено изменять значение параметра. Однако аргумент, передаваемый по значению, можно использовать в теле функции как обычную инициированную переменную: вне функции изменения не видны. Добавление спецификатора const к параметру, передаваемому по значению, не влияет на его интерпретацию. Функции, объявленной как f(int), может быть передано любое значение типа int, равно как и функции f(const int).

Однако, если спецификатор const или volatile применяется к параметру указательного или ссылочного типа, то при сравнении объявлений он учитывается.

```
// объявляются разные функции
void f( int* );
void f( const int* );

// и здесь объявляются разные функции
void f( int& );
```

```
void f( const int& );
```

Все перегруженные функции объявляются в одной и той же области видимости. К примеру, локально объявленная функция не перегружает, а просто скрывает глобальную.

Def. Разрешением перегрузки функции называется процесс выбора той функции из множества перегруженных, которую следует вызвать. Этот процесс основывается на указанных при вызове аргументах.

При разрешении перегрузки функции выполняются следующие шаги:

1. Выделяется множество перегруженных функций для данного вызова, а также свойства списка аргументов, переданных функции.
2. Выбираются те из перегруженных функций, которые могут быть вызваны с данными аргументами, с учетом их количества и типов.
3. Находится функция, которая лучше всего соответствует вызову.

Как правило при исполнении, из нескольких объявлений выбирается та функция, которая наиболее подходит. Проблема: а как решить, что подходит лучше? Правил много, рассмотрим основные принципы:

1. Ищем точное соответствие (exact matching)
2. Promotion от «меньшего» типа к «большему» (пример: short в int)
3. Конвертация из одного типа в другой (пример: int в bool).
4. Определённые пользователем преобразования типа. (Например, пользователь написал конструктор вида String(int[]), в этом случае при необходимости передаваемый в некоторую функцию аргумент типа int[] может быть преобразован к типу String)
5. ellipsis conversion (Функция f(...) имеет самый низкий приоритет, потому что это максимально общий случай. Точно так же функция f(int a, int b) лучше, чем f(int a, ...)).

Общий принцип: от частного к общему. Иногда приходится делать цепочки преобразований.

Замечание: иногда возникает неопределённость, которая может привести к CE. Пример:

```
1  void f(int) {
2      cout << 1;
3  }
4
5  void f(float) {
6      cout << 2;
7  }
8
9  int main() {
10     f(0.0); \\double
11 }
```

И в том, и в другом случае необходима одна конверсия. Компилятор не знает, какую выбрать.

Билет 3.14. Понятие класса, полей и методов класса, модификаторы доступа public и private, отличие класса от структуры. Применение операторов точка и стрелочка. Применение двойного двоеточия. Применение ключевого слова this. Константные и неконстантные методы.

Существует 2 способа создать свой тип.

- Создать свой класс.

```
1 class C {
2 };
3
4 int main() {
5     C c;
6 }
```

  Пустой класс (как на примере) занимает 1 байт в памяти, так как по стандарту C++ никакой объект не может занимать 0 байт в памяти (иначе могло бы так получиться, что какие-то два объекта имеют одинаковый адрес в памяти)

- Создать свою структуру

```
1 struct C {
2 };
3
4 int main() {
5     C c;
6 }
```

  Структуру обычно используют когда не требуется внутренняя логика, нам нужно просто объединить какие-то переменные, если же появляются какие-то методы обработки - используют class.

У классов и структур есть свои поля и методы.

Def. Поля - данные которые хранятся в объекте этого типа, иначе - переменные, объявленные в теле класса.

Def. Методы - операции, которые над ним можно выполнять. Методы можно перегружать, как и обычные функции. Методы можно определить вне класса, если они были объявлены внутри, однако определить метод одного класса внутри другого класса нельзя - у них разные пространства.

Объекты классов бывают константными и неконстантными. Константные объекты класса могут явно вызывать только константные методы класса,

Def. Константный метод — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект) - это делается с помощью квалификатора const. Из константного метода нельзя вызвать неконстантный.

```
class C {
private:
    int s = 0;
    std::string str;
    double d = 0.0;
public:
    void add_and_print(int a);
    void add_and_print(double a) {
        std::cout << d + a;
    }
};
void C::add_and_print(int s) {
    std::cout << C::s + s;
    //std::cout « this->s + s;
}
int main() {
    C c;
    c.add_and_print(1);
}
```

Размер такого класса - сумма размеров всех полей. В целях увеличения производительности иногда к объектам добавляется padding - если в структуре сумма размеров объектов получается больше 8 байт и при этом это число не кратно 8, то компилятор дополняет до числа кратного 8.

Инкапсуляция - первый принцип ООП

Формально: Инкапсуляция - совместное хранение полей и методов (но ограниченный доступ к ним извне).

Неформально: Объявление рядом данных и методов обработки этих данных + ограничение доступа к самим данным. Мы разрешаем доступ извне к данным только разрешенным способам их обработки, т.е пользователь имеет доступ к ограниченному числу методов/полей.

1. private - к этим полям/методам нельзя получить доступ извне.
2. public - к этим полям/методам можно получить доступ извне.

Из main() не получится обратиться к private-полю (оно приватно в этом контексте).
В классе по умолчвнию все поля - private, в структуре же все поля будут public.

Дружественные методы и классы

Иногда нужно все-таки обратиться к private-полям класса не из членов класса, для этого используется ключевое слово **friend**. Нужно внутри класса объявить функцию с этим ключевым словом. В дальнейшем если где-то в коде встретится функция с точно такой же сигнатурой и она не будет членом нашего класса, ей будет разрешен доступ к приватным полям. Если не объявляли что функция может быть методом какого-то класса, то функция с такой же сигнатурой, то в методе какого-то класса не получит доступ к полям. Другом можно объявить не только метод какого-либо класса, но и весь класс - тогда все его методы будут считаться дружественными. Дружба не взаимна и не транзитивна. friend надо использовать в исключительных случаях.

```
class A{
    int s;
    void f(int);
};
class B{
    int t;
}
class String{
    char* str = nullptr;
    size_t size = 0;

    friend void f(int);
    friend void A::g(int);
    friend class B;

};
```

Пусть мы хотим гаранитровать что никто не вызовет функцию от int. Перегрузка выполняется до проверки доступа (найдется точное соответствие), поэтому данный код вызовет ошибку компиляции:

```
class C {
private:
    int s = 0;
    std::string str;
    double d = 0.0;
    void add_and_print(int a) {
        std::cout << s + a;
    }
public:
    void add_and_print(double a) {
        std::cout << d + a;
    }
};

int main() {
    C c;
    c.add_and_print(1);
}

```

Указатель this:

Указатель на текущий объект. В контексте метода класса означает указатель на тот объект, в котором мы сейчас находимся, тот от которого вызван этот метод. This является скрытым первым параметром любого метода класса (кроме статических методов), а типом указателя выступает имя класса. Явно объявить, инициализировать либо изменить указатель this нельзя.

Операторы «.» и «→»:

Оператор «.» – обращение к полю или методу.

Оператор «→» – то же, что и (*p). Обращение по разыменованному указателю объекта к его полям и методам.

Константные и неконстантные методы.

Перегружать функции можно не только по типу параметров функции (по правому операнду), но и по квалификаторам метода (по левому операнду - тому что стоит до точки) - константный/ неконстантный метод. Конструктор, деструктор и не методы класса нельзя помечать как константные/неконстантные.

Напоминание: преобразование неконстантого объекта в константый разрешен, и стоит дешево, а в обратную сторону запрещен.

Следовательно, можно не писать отдельную перегрузку для неконстантного метода, если уже написан константный и они имеют одинаковую логику - при вызоыве такого метода от неконстаного обхекта произойдет неявное преобразование.

Определение метода как константного является частью объявления. Все поля в теле этого метода считаются теперь константными (в т.ч. указатель this) - значит, нельзя применять неконстантые операции к полям или вызывать другие некностантные методы из себя.

Правило: ставить const везде, где метод пригоден для константных объектов.

Кроме того, может понадобиться изменить какие-то поля константного обекта - например если нужно подсчитать сколько раз обратились к методу или для кэширования. Для этого используется ключевое слово **mutable**, которое можно использовать только для полей класса. Mutable это своеобразный anticonst, поле можно будет поменять даже если находимся в константном объекте. Если в полях есть ссылка, тогда даже если метод константый, это поле не будет константным.

Билет 3.15. Понятие конструкторов, деструкторов, пример их правильного определения для класса String. В каких контекстах использования от класса требуется наличие конструктора по умолчанию, в каких - конструктора копирования?

Def. Конструктор - метод, у которого нет возвращаемого значения, который описывает как создать объект класса с заданными параметрами. Как и любой другой метод, его можно определять вне класса. Компилятор может сам создать конструктор по умолчанию, однако в нем будут инициализированы все поля, что плохо в тех случаях, когда среди полей есть указатели.

С C++11 можно делегировать один конструктор другому - сначала выполнится один конструктор, затем другой

```
1    class String{
2        String(...): String(...) {
3        //smth that is need to be done only by second constructor//
4        }
5    };
```

Правило: если в классе определен хотя бы один конструктор, то определение по умолчанию уже не работает.

Если у конструктора есть один или несколько параметров по умолчанию — это по прежнему конструктор по умолчанию. Каждый класс может иметь не более одного кон-

структора по умолчанию: либо без параметров, либо с параметрами, имеющими значения по умолчанию. Например, такой: `MyClass (int i = 0, std :: string s = "");`

В C++ конструкторы по умолчанию имеют существенное значение, поскольку они автоматически вызываются при определенных обстоятельствах, и, следовательно, при определённых условиях класс обязан иметь конструктор по умолчанию, иначе возникнет ошибка:

- Когда объект объявляется без аргументов (например, `MyClass x;`) или создаётся новый экземпляр в памяти (например, `new MyClass;` или `new MyClass();`).
- Когда объявлен массив объектов, например, `MyClass x[10];` или объявлен динамически, например `new MyClass [10]`. Конструктор по умолчанию инициализирует все элементы.
- Когда в классе потомке не указан явно конструктор класса родителя в списке инициализации.
- Когда конструктор класса не вызывает явно конструктор хотя бы одного из своих полей-объектов в списке инициализации.
- В стандартной библиотеке определённые контейнеры заполняют свои значения используя конструкторы по умолчанию, если значение не указано явно. Например, `vector<MyClass>(10);` заполняет вектор десятью элементами, инициализированными конструктором по умолчанию.

Def. Конструктором копирования (англ. copy constructor) называется специальный конструктор, применяемый для создания нового объекта как копии уже существующего. Такой конструктор принимает как минимум один аргумент: ссылку на копируемый объект.

Обычно компилятор автоматически создает конструктор копирования для каждого класса (известные как неявные конструкторы копирования, то есть конструкторы копирования, заданные неявным образом), но в некоторых случаях программист создает конструктор копирования, называемый в таком случае явным конструктором копирования (или «конструктором копирования, заданным явным образом»). В подобных случаях компилятор не создает неявные конструкторы.

Конструктор копирования по умолчанию просто копирует все поля (**shallow copy**) - в т.ч. и указатели, а значит может возникнуть UB - указатели просто перекопировались, но указывают на одну область памяти. Чтобы запретить копирование, можно либо сделать конструктор приватным, либо использовать **delete**.

```
String(const String& s);
```

Если в классе есть поля которые запрещают себя копировать, то дефолтный конструктор копирования не сможет сгенерироваться - выдаст CE.

Существует четыре случая вызова конструктора копирования:

- Когда объект является возвращаемым значением
- Когда объект передается (функции) по значению в качестве аргумента

- Когда объект конструируется на основе другого объекта (того же класса)
- Когда компилятор генерирует временный объект (как в первом и втором случаях выше; как явное преобразование и т. д.)

Конструктор нельзя вызывать как метод от другого конструктора - будет вызывано не от нашей строки, а от копии, которая затем удалится.

Def. Деструктор - метод, который вызывается непосредственно перед тем, как объект будет уничтожен. Как и любой другой метод, его можно определять вне класса.

У деструктора нет параметров, его нельзя перегрузить. Деструктор вызывается когда объект выходит из области видимости. Если в конструкторе не было никаких нетривиальных действий, например выделение памяти или закрытия потоков, то деструктор можно оставить пустым, обнулять какие-то переменные или указатели в деструкторе не нужно, так как после вызова деструктора компилятор снимет эти переменные со стека.

Реализация нескольких конструкторов и деструктора:

```
class String{
private:
    char* str = nullptr;
    size_t sz = 0;
    size_t cpct = 1; \\capacity
public:
    Конструкторы\\
    String() {}
    String(size_t size_, char s = '\0');
    String(const String& other_str);
    Деструктор\\
    String::~String(){
        delete[] str;
    }
};

String::String(size_t size_, char symbol = '\0') : size(size_),
                cpct(size * 2 + 1), str(new char[cpct]){
    memset(str, symbol, size_);
}

String::String(const String& other_str) : size(other_str.size),
                cpct(size * 2 + 1), str(new char[cpct]){
    memcpy(str, other_str.str, size);
}
```

Билет 3.16. Понятие перегрузки операторов, синтаксис перегрузки оператора присваивания. В каких контекстах использования от класса требуется наличие оператора присваивания?

Правило трех: Если в классе потребовалось реализовать нетривильный деструктор, или нетривиальный конструктор копирования или нетриваильный оператор присваива-

ния, то в классе нужно реализовать все три.

Нередко трасформируется в правило пяти:

Правило пяти: Если в классе потребовалось реализовать одного из этих пяти пунктов, стоит реализовывать все.

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

Дефолтный оператор присваивания тоже работает по принципу shallow copy.

Если в классе есть поля, которые не допускают присванивание (например ссылки), то генерация дефолтного оператора присваивания не определена и будет CE (но (!) дефолтный конструктор копирования будет работать).

Операция присваивания копированием отличается от конструктора копирования тем, что должна очищать члены-данные цели присваивания (и правильно обрабатывать самоприсваивание), тогда как конструктор копирования присваивает значения неинициализированным членам-данным.

Про присваивание перемещением:
Определение конструктора перемещения и оператора присваивания перемещением выполняется аналогично определению конструктора копирования и оператора присваивания копированием. Однако, в то время как функции с копированием принимают в качестве параметра константную ссылку l-value, функции с перемещением принимают в качестве параметра неконстантную ссылку r-value.

Теперь всё просто! Вместо выполнения глубокого копирования исходного объекта в неявный объект, мы просто перемещаем (воруем) ресурсы исходного объекта. Под этим подразумевается поверхностное копирование указателя на исходный объект в неявный (временный) объект, а затем присваивание исходному указателю значения null (точнее nullptr) и в конце удаление неявного объекта.
Вот пример реализации оператора присваивания для String.

```
1 String& String::operator =(String other_str){
2     if (this == &s) return *this; //self-assignment
3     swap(other_str);
4     return *this;
5 }
```

Нельзя создавать новые операторы путем перегрузки, можно лишь перегружать существующие, и то не все. Путем перегрузки нельзя поменять приоритет операторов.

Замечание про оператор присаивания.
Если определять оператор + внутри класса, то будет вот такая проблема:

Следующий вызов

```
Complex c(2.0);
1.0 + c;
```

выдаст ошибку, так как мы определили оператор «+» только тогда, когда левым операндом является объект класса (*this).

При определении оператора вне функции и левый, и правый операнд будут равноправны, и компилятор сможет делать каст как левого, так и правого операнда - соответственно в оператор надо передавать два параметра. Тогда корректный код выглядит так:

```
struct Complex{
    double re = 0.0;
    double im = 0.0;

    Complex (const Complex&){}

    Complex& operator +=(const Complex& z) {
        re += z.re;
        in += z.im;
        return *this;
    }
}

Complex operator +(const Complex& a, const Complex& b) {
        Complex copy = a; //Copy constructor definitely called
        return copy += b;
        //copy += b;
        //return copy;
}
```

Билет 3.17. Условия генерации компилятором конструкторов и оператора присваивания. Использование слов default и delete для их запрета или принудительной генерации компилятором.

Ключевое слово **default** введено в C++ 11. Его использование указывает компилятору самостоятельно генерировать соответствующую функцию класса, если таковая не объявлена в классе. Это слово может быть использовано только с конструкторами (по умолчанию, копирования, перемещения), с операторами присваивания, с деструктором. То есть с теми методами, которые компилятор вообще умеет генерировать.

С C++11 можно определять конструктор по умолчанию следующим образом (если поля проинициализированы и среди полей нет ссылок):

```
    String() = default;
```

Ключевое слово **delete** используется для того, чтобы запретить приведение типов или генерацию каких-либо методов. Собственно, это неплохо работает, поскольку сначала производится перегрузка, а потом проверка доступа.

С C++11 можно запретить какой-либо конструктор - т.е нельзя будет вызвать конструктор с некоторыми параметрами:

```
1  class String{
2      String(int n, char c) = delete;
3  };
```

Оператор присваивания по умолчанию почленно присваивает поля. Генерируется только от const type&.

Конструктор копирования по умолчанию просто копирует все поля, не может быть сгенерирован, если какое-то из полей не допускает копирования.

Генерация:

- Конструктор по умолчанию генерируется автоматически, если нет объявленного пользователем конструктора.
- Конструктор копирования генерируется автоматически, если нет объявленного пользователем конструктора перемещения или оператора присваивания перемещением (поскольку в C++03 нет конструкторов перемещения или операторов присваивания перемещением, это упрощается до "always"в C++03).
- Оператор присваивания копированием генерируется автоматически, если нет объявленного пользователем конструктора перемещения или оператора присваивания перемещением.
- Деструктор генерируется автоматически, если нет объявленного пользователем деструктора.

C++11 и позже:

- Конструктор перемещения генерируется автоматически, если нет объявленного пользователем конструктора копирования, оператора присваивания копированием или деструктора, а также если сгенерированный конструктор перемещения действителен.
- Оператор присваивания перемещением генерируется автоматически, если нет объявленного пользователем конструктора копирования, оператора присваивания копированием или деструктора, и если сгенерированный оператор присваивания перемещением действителен (например, если ему не нужно присваивоить постоянные члены).

Билет 3.18. Списки инициализации в конструкторах. Обязательная инициализация полей перед входом в конструктор. Особенности инициализации полей, являющихся ссылками и константами.

Def. 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). Решить это можно с помощью *списков инициализации* в конструкторах.

Синтаксис:

```
struct Integer{
    int field;
    bool complex;
    Integer(int arg, bool complex): field(arg), complex(complex)
    {\\realization}
};
```

Правило : В списке инициализции инициализация происходит не в том порядке, в каком написаны члены списка, а в том, в каком порядке они объявлены как поля класса. Следовательно, в списке инициализии члены нужно перечилсять в таком же порядке, в каком они находятся в полях, иначе может возникнуть UB/CE. Нельзя совмещать списки инициализации и делегирование конструкторов - CE.

```
class A{
    const int& a;
public:
    A(const int& a): a(a){}
}

void f(){
    A a(5);
}
```

Это проблема, потому что 5 - rvalue, и ссыдка на него уничтожится, так что после инициализации будет битая ссылка.

Note. Так как при инициализации внутри конструктора на самом деле происходит копирование полей(это другой namespace), то поля-ссылки и поля-константы должны быть проинициализированы до входа в конструктор. То есть *при их объявлении, либо в списке инициализации.*

Билет 3.19. Понятие наследования, синтаксис объявления наследника, разница между private и public наследованием.

Def. *Наследование* — это одна из основных концепций ООП, позволяющая создавать классы на основе других классов, при этом заимствуя их функционал.

Def. 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.

Def. 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).

Синтаксис наследования:

```
class Base {};
class PublicDerived : public Base {};
class ProtectedDerived : protected Base {};
class PrivateDerived : private Base {};
```

```
class Base {
    int b;
};

class Derived : public Base {
public:
    int a = 5;
    void f(int);
};
```

У класса Derived будут все поля класса Base, плюс свои поля и методы.

Друзья не наследуются. Приватные поля и методы не могут быть унаследованы. В C ++ конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. То есть при создании наследника всегда создается родитель (со всеми полями и т.п.) Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

Сначала поиск метода производится в классе-потомке, а если его там нет, поиск поднимается на ступень выше.

Если в наследнике есть метод, который принимает объект родительского типа, то нельзя будет обратиться к его защищенным полям, но если принимается объект того же класса, что и сам наследник, то к защищенным полям можно обратиться.

```
class Base {
protected:
    int a;
};

class Derived : public Base {
public:
    int a = 5;
    void f(const Base& x) {
        std::cout << x.a; //doesn't work
    void g(const Derived& x) {
        std::cout << x.a; //will work
};
```

У структур по умолчанию наследование публичное, а у классов - приватное.

3.20. Инициализация родителя наследником и запрет преобразования в обратную сторону. То же самое со ссылками и указателями.

```cpp
struct Base {
    int a;
};

struct Derived: public Base {
    int b;
};

int main() {
    Base b;
    Derived d;

    Base b0 = d; // так можно
    Derived d0 = b; // так нельзя

    Base& b1 = d; // так можно
    Derived& d1 = b; // так нельзя, будет CE

    // Чтобы всё-таки привести объект типа Base к типу Derived,
    // нужно воспользоваться static_cast


    Derived& d2 = static_cast<Derived&>(b1);
    // Всё пройдет успешно, так как под b1 на самом деле лежит объект типа Derived

    Derived& d3 = static_cast<Derived&>(b);
    // Никакой ошибки не выведется, однако будет UB, так как под b на самом деле не лежит объект типа Derived

    Base* b4 = &d;
    Derived* d4 = static_cast<Derived*>(b4);
    // так можно

    Derived* d5 = &b;
    Base* b6 = static_cast<Base*>(d5);
    // так нельзя
    return 0;

}
```

3.21. Понятие полиморфизма. Понятие виртуальных функций. Разница в поведении виртуальных и невиртуальных функций при наследовании. Предназначение и использование ключевых слов override и final.

Идея: Пусть у нас есть геометрические фигуры. Площадь квадрата человек будет считать как произведение сторон, площадь прямоугольного треугольника - половина произведения катетов, площадь произвольного многоугольника - будет триангулировать и считать площадь. Цель методов одна и та же (найти площадь), реализация - разная.

Полиморфизм - один из главных столпов объектно-ориентированного программирования. Его суть заключается в том, что один фрагмент кода может работать с разными типами данных. Свойство, которое позволяет использовать одно и тоже имя функции для решения двух и более схожих, но технически разных задач.

```
// =============== Версии без полиморфизма ===================

struct Base_NoPolymorphism {
  void f() { cout << 1; }
};
struct Derived_NoPolymorphism: public Base_NoPolymorphism {
  void f() { cout << 2; }
};

// =============== Версии с полиморфизмом ===================

struct Base {
  virtual void f() { cout << 1; } // (1)
};

struct Derived: public Base {
  // void f() const  cout « 2;  - не виртуальный, т.к. не совпадает с (1)
  // virtual void f() const  cout « 2;  - виртуальный, но не переписывает (1)
  virtual void f() {cout << 2; }
  int check = 5;
};

int main () {
  // До полиморфизма:
  Base_NoPolymorphism b;
  b.f(); //1
  Derived_NoPolymorphism d;
  d.f(); //2
  Base_NoPolymorphism bb = d;
  bb.f(); //1
  Base_NoPolymorphism& bbb = d;
  bbb.f(); //(*), печатается 1, взяли версию родителя

  // С полиморфизмом:
  Base b1;
  b1.f(); //1
  Derived d1;
  d1.f(); //2
  Base bb1 = d1;
  bb1.f(); // 1 - мы считаем этот объект родителем
  Base& bbb1 = d1;
  bbb1.f(); // 2 - он "помнит каким ребёнком был d1
}
```

Виртуальная функция - такая функция, что если обратиться к ней вне зависимости от того, это объект общего типа или частного типа, вы предпочитаете реализацию в объекте частного типа.

Для виртуальных функций выбор происходит в Runtime, для не виртуальных в Compile time. Именно поэтому в примерах выше если мы делаем копию, то выбирается родительский метод (всё определилось в Compile Time), а для ссылок/указателей всё выбирается в Runtime, мы выбрали дочерний метод.

Виртуальная функция дочернего класса является переопределением только если совпадают её сигнатура и тип возврата с сигнатурой и типом возврата виртуальной функции родительского класса. Для проверки того, что данная функция является переопределением, добавили модификатор **override** в C++11, из-за которого компилятор выдаёт CE, если метод не переопределяет виртуальную функцию родительского класса.

```
struct A {
  virtual const char* getName(int x) { return "A"; }
};

struct B : public A {
  // virtual const char* getName(short int x) override  return "B";  - CE
  // virtual const char* getName2(int x) const override  return "B";  - CE
  virtual const char* getName3(int x) override { return "B"; } // OK
};
```

Модификатор **final** используется, если вы не хотите, чтобы кто-то мог переопределить виртуальную функцию или наследовать определенный класс. Если пользователь пытается переопределить метод или наследовать класс с модификатором final, то компилятор выдаст ошибку.

```
struct A {
  virtual const char *getName() { return "A"; }
};

struct B : public A {
  virtual const char *getName() final { return "B"; } // OK
};

struct C : public B {
  virtual const char * getName() { return "C"; } // CE
};
```

NOTE: для любой функции, которая является "дочерней"от какой-то родительской (помеченной virtual) формально можно не писать virtual, если у неё нет своих "детей". Однако на практике часто проще ориентироваться в виртуальных функциях, если это подписано всегда.

3.22 Понятия шаблонов, инстанцирования шаблонов, специализации шаблонов. Синтаксис определения шаблонов классов и шаблонов функций. Синтаксис определения специализации шаблонов

Определение *Шаблоны* — это средство языка, предназначенное для написания кода без привязки к конкретному типу.
Определение *Инстанцирование шаблона* — процесс создания конкретного класса/функции/using из шаблона путем подстановки аргументов. Процесс инстанцирования шаблонов происходит между препроцессингом и компиляцией. Происходит с проверкой типов аргументов (совместимость по присваиванию, по приведению типов, по вызываемым методам). Для классов нужно явное инстанцирование
Определение *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.
Что можно определить, как шаблон? Функцию, класс, псевдоним, переменную, метод класса

```
template <typename T, typename U>
T maximum(const T& a, const U& b) {
    return a > b ? a : b;
}

template <typename T>
class vector {

};

// since c++11
template <typename T>
using mymap = std::map<T, T>;

// since c++14
template <typename T>
const T pi = 3.14;
```

```
template <typename T>
struct vector {
    template <typename U>
    void push_back(const U&);
};

template <typename T>
template <typename U>
void vector<T>::push_back(const U& x) {
    // ...
}
```

Пример специализации шаблонов для классов

```
template <typename T>
class vector {

};

// full specialization.
template <>
class vector<bool> {

};
```

Это полная специализация. Создаем частный случай шаблонного класса. При этом новый и старый классы могут вообще не совпадать в реализации.

```cpp
// partial specialization.
template <typename T>
struct S<T*> {
    void f() {
        std::cout << 3;
    }
};
```

Частичная специализация. определенная для всех типов, являющихся укзателями

Пример специализации шаблонов для функций

Для функций частичная специализация запрещена стандартом, можно делать только полную

```cpp
template <typename T, typename U>
void f(T, U) {
    std::cout << 1;
}

template <typename T>
void f(T, T) {
    std::cout << 2;
}

template <>
void f(int, int) {
    std::cout << 3;
}
```

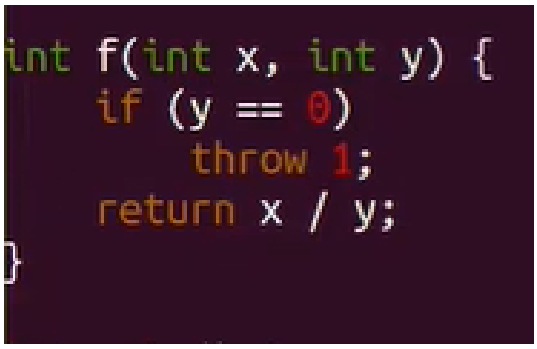
Функция 3 - специализация 2 функции

```cpp
template <typename T, typename U>
void f(T, U) {
    std::cout << 1;
}

template <>
void f(int, int) {
    std::cout << 3;
}

template <typename T>
void f(T, T) {
    std::cout << 2;
}
```

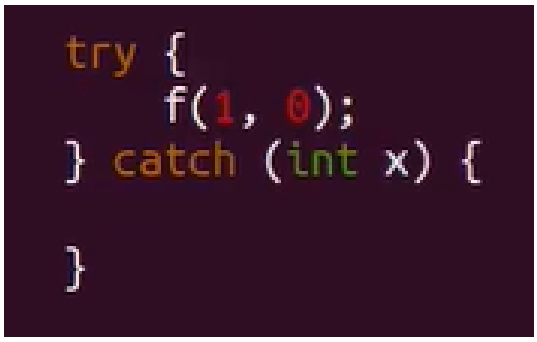
Функция 3 - специализация 1 функции

3.23 Понятие исключений. Синтаксис использования throw и конструкции try... catch. Что является и что не является исключениями. Пример ошибки времени выполнения, не являющейся исключением.

Определение *Исключение* — нестандартная ситуация, возникающая в ходе выполнения программы.
В плюсах можно бросать что угодно. Например, int



Try...catch



Try может поймать только то, что с помощью оператора throw брошено (только тот вид ошибки, который мы указали, только на том уровне, на котором мы хотим) Например, если мы вызвали из try ... catch в main какую-то функцию, эта функция запустила еще функцию, а та еще десять, и в какой-то из них сработало исключение, это исключение полетит наверх до main, и если оно не поймалось, выдаст ошибку

Если ошибка не смогла скастоваться к тому виду ошибок, которые мы ловим, она не поймается :(

Разница между runtime ошибками и исключениями

Не любой runtime error является исключением. Через try ... catch можно отловить только то, что брошено throw, и только это

- × Деление на ноль - не исключение
- × UB - не исключение
- × delete по плохому указателю - не исключение
- × Битая ссылка - не исключение
- × Выход за границу массива - не исключение

Выход за границу массива

```
std::vector<int> v(10);
try {
    v[1000000] = 1;
} catch (...) {
    std::cout << "!!!";
}
```

Выдаст seg fault

Конструкция catch(...) означает, что поймается что угодно

3.24 Наличие/отсутствие и алгоритмическая сложность различных операций в контейнерах

Первые три контейнера - последовательные (sequence containers), вторые - associative containers.

Container	indexating [*pos*]	push_back	insert(it)	erase(it)	find	iter
vector	$O(1)$	$O(1)$ amort	$O(n)$	$O(n)$	-	RA
deque	$O(1)$	$O(1)$	$O(n)$	$O(n)$	-	RA
list (forward_list)	-	$O(1)$	$O(1)$	$O(1)$	-	BI (FI)
set/map	$O(logn)$	-	$O(logn)$	$O(logn)$	$O(logn)$	BI
unordered_set/map	$O(1)$*	-	$O(1)$*	$O(1)$*	$O(1)$*	FI

Пояснение: $O(1)$* - это $O(1)$ среднее, т.е такое, что можно подборать набор входных данных что операции будут работать за линию, но в среднем операции работатют за $O(1)$, так как хеширование реализованно методом цепочек.
Замечание: Почему в deque push_back за $O(1)$? Память двухсторонней очереди автоматически расширяется и сокращается по мере необходимости. Расширение двухсторонней очереди дешевле, чем расширение std::vector, поскольку оно не включает в себя копирование существующих элементов в новое расположение в памяти. С другой стороны, двухсторонние очереди обычно имеют большую минимальную стоимость памяти; двухсторонняя очередь, содержащая только один элемент, должна выделить свой полный внутренний массив (например из 8 ячеек).
!!! Заметим, что у **forward_list, list** и **deque** есть метод `push_front`, который работает за $O(1)$.

3.25 Основные методы контейнера vector и их правильное применение

reference operator[](size_type pos); Возвращает ссылку на элемент по индексу pos. Проверка на выход за границы не выполняется. В методе **at(pos)** эта проверка есть.

void push_back(const T& value); Добавляет данный элемент value до конца контейнера. Если новый size() больше, чем capacity(), Все итераторы и указатели становятся нерабочими. В противном случае, все они остаются в рабочем состоянии. При этом value должен быть CopyInsertable или MoveInsertable

void pop_back(); Удаляет последний элемент контейнера. Итераторы и указатели остаются в рабочем состоянии.

size_type size() const; Возвращает количество элементов в контейнере, т.е.
`std::distance(begin(), end())`

void resize(size_type count, const value_type& value); Изменяет размер контейнера, чтобы содержать count элементы. Если текущий размер меньше, чем count, дополнительные элементы добавляются и инициализируется value (или конструируются по умолчанию). Если текущий размер больше count, контейнер сводится к ее первые элементы count.

size_type capacity() const; Возвращает количество элементов, для которого сейчас выделена память контейнером.

void reserve(size_type size); Задает емкость контейнера по крайней мере, size. Новая память выделяется при необходимости.

3.26 Основные методы контейнера map и их правильное применение

T& operator[](const Key& key); Вставляет новый элемент в контейнере с помощью key в качестве ключа и конструктором по умолчанию отображенное значения и возвращает ссылку на вновь построенное значение. Если элемент с ключом key уже существует, вставка не выполняется, и возвращается ссылка на это значение. Итераторы и указатели остаются в рабочем состоянии.

T& at(const Key& key); Возвращает ссылку на соответствующее значение элемента с ключом, эквивалентным key. Если такого элемента не существует, бросает исключение типа `std::out_of_range`.

iterator find(const Key& key); Находит элемент с ключом key и возвращает итератор на этот элемент. Если такой элемент не найден, то возвращается end()

std::pair<iterator,bool> insert(const value_type& value); Вставляет элемент value в контейнер, если контейнер еще не содержит элемент с эквивалентным ключом. Возвращает пару из итератора на вставленный элемент (или на тот, который помешал вставке), и bool, указывающий, была ли вставка.

void insert(InputIt first, InputIt last); Вставляет элементы из диапазона [first, last). При этом диапозон задан Input Iterator'ами. Ничего не возвращает.

void insert(std::initializer_list<value_type> ilist); Вставляет список инициализации `std::initializer_list<value_type>` и ничего не возвращает.

iterator erase(const_iterator position); Удаляет из контейнера элемент на позиции pos. Указатели и итераторы к удалённым элементам становятся недействительными. Другие итераторы и указатели остаются без изменений. Возвращает итератор, следующего элемента за удаленным элементом.

iterator erase(const_iterator first, const_iterator last); Удаляет элементы из контейнера в диапазоне [first; last). Указатели и итераторы к удалённым элементам становятся недействительными. Другие итераторы и указатели остаются без изменений. Возвращает итератор, следующего элемента за удаленным элементом.

iterator lower_bound(const Key& key); Возвращает итератор, указывающий на первый элемент, который является *не меньше, чем key*. Если такой элемент не найден, то возвращается end()

iterator upper_bound(const Key& key); Возвращает итератор, указывающий на первый элемент, который является *не больше, чем key*. Если такой элемент не найден, то возвращается end()

3.27 Категории итераторов

Input Iterator: позволяет лишь раз пройтись по последовательности.

- Копирование, присваивание.
- Операции сравнения на равенство `it1 == it2` и `it1 != it2`
- Инкремент: `++it` и `it++`.
- Разыменование для чтения: `*it` и `it->m`,
  при этом **запрещена** запись: `*it = value;`

Пример входного итератора - это итератор чтения из потока: `std::istream_iterator`

Важно: Если его скопировать и пройтись 2ой раз, не гарантируется что получим ту же последовательность.

Forward Iterator: однонаправленные итераторы, могут перемещаться только в одну сторону на 1 позицию, перемещение в обратную сторону занимает продолжительное время.

- Все операции **Input Iterator**
- Разыменование для записи: `*it = value;` и `*it++ = value;`
- Требование *многопроходности*: если `it1 == it2`, то `++it1 == ++it2`, т.е. итератор можно копировать, и обходить им последовательность много раз.

Например, итераторы **forward_list, unordered_map, unordered_set**

Bidirectional Iterator двунаправленные итераторы, могут быстро перемещаться на одну позицию как вперед, так и назад.

- Все операции **Forward Iterator**
- Декремент: `--it, it--, *it--`

Например, итераторы **list, map, set**

Random-access итераторы: могут перемещаться быстро на любую позицию в контейнере.

- Все операции **Bidirectional Iterator**
- Операции сравнения: `it1 < it2, it1 > it2, it1 <= it2, it1 >= it2`
- Сложение/вычитание с числом: `it + n, it += n, it - n, it -= n`
- Разность итераторов: `it2 - it1`
- Индексирование: `it[n]`

Например, итераторы **vector, deque**

Алгоритмы из STL, реализованные с помощью итераторов :

- InputIterator - find()
- ForwardIterator - binary_search() т.е. определяет, находится ли элемент в некотором отсортированном диапазоне
- BidirectionalIterator - next_permutation() т.е. генерирует ближайшую следующую перестановку диапазона элементов
- Random-Access Iterator - sort()

3.28 Использование итераторов для прохода по любому из контейнеров от начала до конца. Использование range-based for.

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

//access by operator []
for(int i = 0; i < v.size(); ++i)
    std::cout << v[i];

//access by iterator
for(std::vector<int>::iterator it = v.begin(); it < v.end(); ++it)
    std::cout << *it;
```

Начиная с C++11 есть синтаксис **range-based for:**

1. Для **просмотра значений** используется такой синтаксис:

```
for (const auto& elem : container)    // capture by const reference
```

   - Если объекты *дешевые для копирования* (например, int, double, etc), то можно использовать немного упрощенную форму:

```
for (auto elem : container)    // capture by value
```

     В цикле используется локальная копия объекта из контейнера.

2. Для **изменения значений** в контейнере используется такой синтаксис:

```
for (auto& elem : container)    // capture by (non-const) reference
```

   - Если контейнер использует так называемые *proxy iterators* (например, std::vector<bool>), то нужно использовать такую форму:

```
for (auto&& elem : container)    // capture by rvalue reference
```

     Объясним почему именно так. Если запустить код, который инвертирует значения булевского массива, то мы получим CE:

```
vector<bool> v = {true, false, false, true};
for (auto& x : v)  // CE
    x = !x;
```

     Проблема в том, что std::vector специальизированный для bool реализован с оптимизацией памяти (то есть каждое логическое значение лежит в 1 бите $\Rightarrow$ 8 логический значений в одном блоке вектора). Так как невозможно вернуть ссылку на каждый бит, вектор использует *proxy iterators*, который возвращает по значению временный объект, а именно proxy class convertible to bool. Поэтому корректная реализация такая:

```
vector<bool> v = {true, false, false, true};
for (auto&& x : v)  // OK
    x = !x;
```

Вывод – универсальный подход такой:

```
// For observing the elements, use:
for (const auto& elem : container)

// For modifying the elements in place, use:
for (auto&& elem : container)
```

3.29. Использование стандартных алгоритмов STL над контейнерами: std::sort, std:find, std::find_if, std::max_element, std::min_element, std::reverse, std::binary_search, std::lower_bou std::upper_bound, в том числе с произвольной функцией сравнения.

std::sort

std::sort применяется над контейнерами с итераторами категории Random Access Iterator для того, чтобы отсортировать их элементы на полуинтервале [first, last) за $O(nlogn)$, где n — количество элементов, first, last — итераторы. По умолчанию элементы сортируются по возрастанию, с применением произвольного компаратора этот порядок возможно изменить.

```
#include <iostream>
#include <algorithm>
#include <vector>

template<typename T>
struct myComp {
    bool operator()(T& a, T& b) {
        return a > b;
    }
};

int main() {
    std::vector<int> theormax = {10, 2, 3, 4};
    std::sort(theormax.begin(), theormax.end()); //[2, 3, 4, 10]
    std::sort(theormax.begin(), theormax.end(), myComp<int>()); //[10, 4, 3,
   2]
    return 0;
}
```

std::find, std::find_if

std::find используется для поиска конкретного элемента в контейнере: если он есть, то возвращается итератор на него, иначе на итератор end. std::find_if используется аналогичным образом для поиска элемента, отвечающего заданному условию.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> v{1, 2, 3, 4};
    int n1 = 3;
    int n2 = 5;
    auto is_even = [](int i){ return i%2 == 0; };

    auto result1 = std::find(begin(v), end(v), n1);
    auto result2 = std::find(begin(v), end(v), n2);
    auto result3 = std::find_if(begin(v), end(v), is_even);

    (result1 != std::end(v))
        ? std::cout << "v contains " << n1 << '\n'
        : std::cout << "v does not contain " << n1 << '\n';
    //v contains 3

```

```
    (result2 != std::end(v))
        ? std::cout << "v contains " << n2 << '\n'
        : std::cout << "v does not contain " << n2 << '\n';
    //v does not contain 5

    (result3 != std::end(v))
        ? std::cout << "v contains an even number: " << *result3 << '\n'
        : std::cout << "v does not contain even numbers\n";
    //v contains an even number: 2
}
```

std::max_element, std::min_element

std::max_element находит наибольшее значение в контейнере на полуинтервале [first, last), где first, last — итераторы категории не ниже Forward Iterator. Аналогичным образом находит наименьшее значение std::min_element.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>

static bool abs_compare(int a, int b) {
    return (std::abs(a) < std::abs(b));
}

int main() {
    std::vector<int> v{ 3, 1, -14, 1, 5, 9 };
    std::vector<int>::iterator result;

    result = std::max_element(v.begin(), v.end());
    std::cout << "max element at: " <<
    std::distance(v.begin(), result) << '\n'; // 5

    result = std::max_element(v.begin(), v.end(), abs_compare);
    std::cout << "max element (absolute) at: " <<
    std::distance(v.begin(), result) << '\n'; // 2
}
```

std::reverse

std::reverse перемещает элементы в обратном порядке относительно их исходного расположения на полуинтервале [first, last), где first, last — итераторы категории не ниже Bidirectional Iterator.

std::binary_search, std::lower_bound, std::upper_bound

std::binary_search методом бинарного поиска (если поддерживается Random Access Iterator) пытается найти данный элемент на [first, last), если нашлось, возвращается true, иначе false.

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> haystack {1, 3, 4, 5, 9};
    std::vector<int> needles {1, 2, 3};

    for (auto needle : needles) {
        std::cout << "Searching for " << needle << '\n';
        if (std::binary_search(haystack.begin(), haystack.end(), needle)) {
            std::cout << "Found " << needle << '\n';
        } else {
```

```
            std::cout << "no dice!\n";
        }
    }
}
//Searching for 1
//Found 1
//Searching for 2
//no dice!
//Searching for 3
//Found 3
```

std::lower_bound ищет первый элемент, не меньший данного, на [first, last), std::upper_bound ищет первый элемент, больший данного, на [first, last). В случае успеха возвращается итератор на найденный элемент, иначе возвращается итератор на end.

3.30 Мотивировка move-семантики. Какую проблему она решает? Пример ситуации, когда уместно использовать std::move. Пример правильной реализации move-конструктора и move-оператора присваивания для класса String.

Мотивировка move-семантики:

1. Когда мы хотим передать объект таким образом, мы вынуждены идти на одно "одноразовое"копирование

```
int main() {
    vector<string> v;
    v.push_back(string("abc"));
}
```

2. Далее есть реалокация при увеличении размера вектора, когда старый заполнился. А перекладывание - это снова $new(newarr+i)T(arr[i])$. Если в векторе лежали строки, то по факту - вектор просто хранил указатели на динамическую память каждой строки, тогда вопрос: зачем нам копировать все строки, если можно было просто переставить указатели (но напрямую так сделать нельзя)

3. Рассмотрим реализацию функции swap.

```
template <typename T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = t;
}
```

   В данном коде наблюдается тройное копирование в случае сложных объектов.

4. Пусть есть функция, результатом которой является новый объект типа T.

   (a) При вызове этой функции в другой части кода MyHeavyType object = createObject(); - будет все ок, так как произойдет Copy Elision (оптимизация компилятора).

   (b) Но если мы делаем f(createObject()), то мы не можем принять объект в f по значению, так как точно произойдет копирование (вызовется конструктор копирования). Если принять объект по константной ссылке, то мы не сможем менять его.

```
template <typename T>
T createObject(...) {
    //...
    return obj;
}

int main() {
    vector<string> v;
    v.push_back(string("abc"));
    //получаем лишнее копирование
    foo(createObject());
}
```

Пример использования:

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

После того, как объект мувают, все его параметры возращаются к дефолтным, например, размер мувнутой строки равен нулю. **К объекту, который мувнули, можно обращаться, гарантируется, что он останется в валидном состоянии.**

Если же объект мувнули и результат действия нигде не используется, то объект остаётся нетронутым:

```
int main() {
    string s("abc");
    std::move(s);
}
```

Пример правильной реализации move-конструктора и move-оператора присваивания для класса String

```
String(String&& s): sz(s.sz), s(s.str) {
    s.str = nullptr;
    s.sz = 0;
}

String& operator=(String&& s) {
    String temp = std::move(s);
    swap(temp);
    return *this;
}
```

3.31 Мотивировка умных указателей. Какую проблему они решают? Базовый синтаксис использования shared_ptr (как создать, как пользоваться в простейшем случае)

см. билет 4.28.

4.1 Параметры компилятора и объяснение их действия. Примеры предупреждений компилятора, не являющихся ошибками с точки зрения языка. Пример неожиданного поведения компилятора при UB с переполнением int и включенном параметре -O. Объяснение этого поведения.

Параметры компилятора:

- **-Wall** Включает базовые предупреждения, в том числе предупреждения, отключенные по умолчанию. Например:
  * **-Wreorder** Предупреждает о том, что порядок инициализации членов класса не соответствует порядку их объявления. Поскольку компилятор может переупорядочить инициализацию этих членов, результат может быть неочевидным.
  * **-Wreturn-type** Предупреждает о том, что из функции не вернули заявленный результат
  * **-Wunused-variable** Предупреждает о том, что переменная не используется.
- **-Wextra** Включает дополнительные предупреждения, которых нет в -Wall
  * **-Wempty-body** Предупреждает о пустом теле условных выражений
  * **-Wunused-parameter** Предупреждает о неиспользуемом параметре функции. Возможно, про него забыли, и в этом случае функция может работать некорректно
  * **-Wmissing-field-initializers** Предупреждает о том, что отдельные члены структуры не были проинициализированы. Скорее всего это просто забыли сделать
- **-Werror** Сообщает компилятору, чтобы все предупреждения были превращены в ошибки, и при их наличии компиляция прерывалась.
- **-O1, -O2, -O3** Различные уровни оптимизации.
- **-O0** Отключение оптимизации.
- **-std=c++11, -std=c++14, -std=c++17, -std=c++2a** Подключение функционала C++11/14/17/20 соответственно.

Примеры предупреждений компилятора, не являющихся ошибками с точки зрения языка:

```
    // assignment in a conditional statement
    int x=3, y=4;
    if(x=y) {}

    // implicit type conversion
    int n = 10;
    for(size_t i = 0; i < n; ++i){}
```

Подробнее про все предупреждения можно почитать тут: ссылочка на хабр

Интересное UB с параметром -O2

Рассмотрим пример того, как неопределенное поведение в программе может приводить к неожиданным последствиям. Обратимся к коду ниже:

```
    for(int i = 0; i < 300; i++) {
        cout << i << " " << i * 12345678 << endl;
    }
```

Если скомпилировать этот код без параметра оптимизации, то мы получим, просто 300 чисел (при этом на 174 шаге происходит переполнение и выводятся отрицательные числа). Однако, если скомпилировать данный код с оптимизатором -O2, то цикл станет бесконечным. Почему? Компилятор считает, что ввод корректен (прогер не дурак), значит i не превосходит 173 (так как иначе происходит переполнение), поэтому оптимизатор заменяет условие `i < 300` на true и бинго, у нас бесконечный цикл.

Билет 4.2. Разница между понятиями operator precedence и order of evaluation. Понятие unspecified behaviour, примеры. Несколько примеров выражений, вычисление которых является unspecified behaviour (с объяснением). Примеры выражений, вычисление которых является undefined behaviour.

Operator precedence vs. order of evaluation

На порядок вычисления значений функций, в отличие от порядка выполнения операций, стандартом C++ никаких требований не наложено, и порядок вычисления значений функций является неспецифицированным.

Unspecified behaviour, примеры

Def. Unspecified behaviour — неустановленное поведение, поведение для корректной программы и данных, зависящее от компилятора.

Пример:

```
#include <iostream>

int f() {
    std::cout << 1;
    return 2;
}

int g() {
    std::cout << 2;
    return 3;
}

int main() {
    f() + g();
    return 0;
}
```

Выведется 12, ясно, что нужно сложить f() + g(), но стандарт не предписывает, что именно нужно сначала вычислить: f() или g().

Аналогичные примеры: f() + g() * g(), f(g(), h()).

Один из примеров выражений, приводящих к UB

```
int i = 55;
i = ++i + i++;
```

Билет 4.3. Приведения типов: *C − style cast*, *static_cast*, *const_cast* и *reinterpret_cast*. В каких ситуациях (кроме наследования) каждый из этих операторов приводит к ошибке компиляции, в каких - к неопределенному поведению, а в каких он уместен?

static cast: принимает решение на этапе компиляции и может, в частности,

- Вызвать конструктор или определённый пользователем оператор преобразования типа — в частности, помеченный как explicit
- Преобразовать тип указателя или ссылки в случае наследования и ряде других
- Использовать стандартное преобразование типа.

Это стандартное преобразование между типами, которое вы ожидаете. В частности указатели несовместимых типов, как например int и double, не переводятся друг в друга — CE, а переполнение — это UB.

```
int x = 42;
*static_cast<double*>(&x); //CE
long long y = 24;
static_cast<int>(y); //UB
```

reinterpret cast: "более топорно меняет тип выражения. Не выполняет никаких дополнительных операций в рантайме. Разрешаются любые преобразования указателей, не понижающие константность."

Эта штука тупо считает кусок памяти начинающийся с некоторого момента куском памяти другого типа, в частности если int кастануть к double и спросить что там лежит — будет UB, так как залезли не в свою память. Есть легкая и тяжелая форма каста: через указатель и через ссылку соответственно.

Легкая:

```
int x = 42;
std::cout << *reinterpret_cast<double*>(&x); //UB
```

Тяжелая:

```
double d = 3.14;
std::cout << std::hex << reinterpret_cast<int&>(d); //output  eb51851f побитоое расположение в памяти
```

Тут мы урезали кусок памяти с дабла до инта. Тяжелый каст — первое запрщенное заклинание курса, так как он слишком опасен.

```
double d = 3.14;
reinterpret_cast<int>(d); //CE
```

```
const int &const_iref = i;
//int &iref = reinterpret_cast<int&>(const_iref); //compiler error - can't get rid of const
//Must use const_cast instead: int &iref = const_cast<int&>(const_iref);
```

const_cast: говорим компилятору, что на самом деле объект, который мы хотим поменять неконстантый, и если он действительно такой, то компилятор пускает

```
int x = 5;
const int& y = x;
int& z = const_cast<int&>(y);
z = 10;
std::cout << y; //cout « 10
```

а если нет, то это UB

```
const int cx = 1;
int& x = const_cast<int&>(cx);
x = 2;
std::cout << cx; //cout « 1; UB
```

Вроде как такой эффект достигается тем, что константы лежат где-то в специальной части памяти или же тут же подставляются, но мы, вроде, этого не обсуждали на лекциях.

```
double d = 3.14;
const_cast<double>(d); //CE
```

```
[[maybe_unused]]
void (type::* pmf)(int) const = &type::f; // pointer to member function
// const_cast<void(type::*)(int)>(pmf); // compile error: const_cast does
                                        // not work on function pointer
```

C-style cast: жуткий каст, который пытается использовать вспе подряд, пока не сработает С лекции:"Стандарт говорит: сначала пытается сделать const cast, если не сработает, то пытается сделать static, если опять нет, то пытается как static + const, затем пытается reinterpret, затем reinterpret + const. Поэтому если даже он сработает — вы не будете знать как именно"

```
int x = 0;
double d = (double)x;
long long y;
(int)y; //UB, так как отработает как static_cast
struct A{};
struct B{};
B b;
(A)b; //CE, так как ни один из кастов не возможен
```

Билет 4.4. Статические переменные, особенности и примеры их использования. Статические поля и методы классов, статические константы, их особенности и примеры использования. Понятие статической памяти, разница между статической и стековой памятью.

Статические переменные: Использование ключевого слова static с локальными переменными изменяет их свойство продолжительности жизни с автоматического на статическое (или «фиксированное»). Статическая переменная (или «переменная со статической продолжительностью жизни») сохраняет свое значение даже после выхода из блока, в котором она определена. То есть она создается (и инициализируется) только один раз, а затем сохраняется на протяжении выполнения всей программы.

```
#include <iostream>

void incrementAndPrint()
{
    static int s_value = 1; \\ переменная s_value является статической
    ++s_value;
    std::cout << s_value << std::endl;
} \\ переменная s_value не уничтожается здесь, но становится недоступной

int main()
{
    incrementAndPrint(); \\ cout << 2
    incrementAndPrint(); \\ cout << 3
    incrementAndPrint(); \\ cout << 4
}
```

Статические поля и методы: поля и методы обозначеные словом static, являются общими для всего класса, а не отдельного объекта, при том выделяются в статической памяти.

```
struct A {
    static int a;
};

int A::a = 1;

int main() {
    A a;
    std::cout << a.a++ << A::a++ << A().a++; \\ 123
}
```

Для метода аналогично. Примером их использования может быть счетчик количества объектов данного класса. Или синглтон. Из статических методов нельзя обращаться к нестатическим полям класса. Статический переменные можно инициализировать внутри класса, только если они константные и целочисленные. Выделяются статические переменные при запуске программы в статической памяти.

Статическая память: память выделяющаяся в начале работы программы, которая содержит статические переменные и поля, а также глобальные переменные. При запуске программы упрощенно компьютер выделяет память:

[DATA | TEXT | STACK - 4(8)MB]

На стеке выделяется обычно 4 или 8 мегабайт. DATA хранит статические переменные, там выделяется статическая память. TEXT содержит код программы. Стэк хранит локальные переменные, а также указатели на запускаемые функции, то есть рекурсия происходит на стеке. Потому может произойти Stack Overflow — переполнение стека, например, при рекурсивном запуске функции из самой себя много раз подряд (Segmentation Fault).

Билет 4.5. Приватные и публичные поля и методы. Особенности выбора версии функции при перегрузке в случае выбора между приватными и публичными версиями (с примерами). Функции-друзья и классы-друзья, синтаксис объявления, пример использования: перегрузка операторов ввода-вывода. Свойства отношения дружбы в C++.

Публичные поля и методы доступны всем. Приватные поля и методы доступны только классу и его друзьям. Для понимания выбора версии функции важно запонмить золотое правило: Перегрузка выполняется до того как происходит проверка доступа. Проверка доступа — в самый последний момент.

Приватное поле, к которому нет доступа, не то же самое, что поля вообще нет. visible не равно accessible. Видимые – те, которые находит поиск имен. Доступные – те, к которым есть доступ по модификаторам доступа при наследовании.

```
1  struct Granny{
2      void f(){
3          std::cout << "Granny";
4      }
5  };
6
7  struct Mom : private Granny{
8      void f(int y){
9          std::cout << "Mom";
10     }
11 };
12
13 struct Son : Mom {
14     void f(double y){};
15 };
16
17 int main(){
18     Mom m;
19     m.f(); // output: "Mom"
20     Son s;
21     //s.f(); //CE, this func is inaccesable
22     //m.Granny::f(); // CE, as Granny inaccessible
23     s.Mom::f(0);
24 }
```

При вызове метода f от объекта типа Mom вызовется метод из функции Mom, так как этот метод, будучи названным так же, как и метод наследуемого класса, "перекрывает"функцию f из Granny, f из Granny - not visible. Поля и методы с одинаковыми именами в классе-наследнике более локальные, чем в классе родителя. Поля класса-родителя перекрываются и не видны из класса наследника.

Доступность проверяется после разрешения перегрузки и выбора версий. В данной версии бабушкина версия не видна, а мамина недоступна - CE:

```
struct Granny{
    void f(){
        std::cout << "Granny";
    }
    void g(double);
};

struct Mom : Granny{
private:
    void f(){
        std::cout << "Mom";
    }
    void g(int);
};

int main(){
    Mom m;
    m.f();
    m.g(0.0);// conversion to int, bc even if Grannies candidate is perfect, it is not visible
    m.Granny::f(0.0) // will work
    std::cout << m.a;

}
```

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). Дружба не наследуется.

```
class String {
public:
    String(const char[]);
    // позволяет определить оператор вывода в поток
    friend std::ostream& operator<<(std::ostream& out, const String& s);
    friend std::istream& operator>>(std::istream& in, 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;
```

```
}

std::istream& operator>>(std::istream& in, BigInteger& number) {
    std::string input;
    in >> input;
    // ...длинный код преобразования строки ввода input в BigInteger...
    return in;
}
```

Билет 4.6. Ключевое слово explicit, два возможных контекста его использования. Перегрузка операторов приведения типа для классов, пример. Определение пользовательских литеральных суффиксов для классов, пример.

Ключ-слово explicit — запрещают неявную конверсию типов. Использовать его можно в двух ситуациях: в конструкторах и операторах привидения типа.

```
struct UserId {
    int id = 0;
    explicit UserId(int x): id(x) {}

    explicit operator int() { // тип возвращаемого значения указывать не нужно
        // возвращаем число по UserId
    }
};

int main{
    UserId u = 5; //CE
    UserId u2(5);

    return 0;
}
```

Типичный пример перегрузки операторов приведения типа — operator bool() в BigInteger: при записи f(x) подстановка считается явной (контекстуальной конверсией). Тогда этот оператор просто проверяте, что return (BigInteger != 0);

Перегрузка литеральных суффиксов, это когда вы перегрузили 5_uid и теперь, если вы где-то в коде напишите 5_uid, то это будет UserId с id = 5; (Нижнее подчеркивание — часть синтаксиса)

```
UserId operator ""_uid(unsigned long long x) {
    return UserId(5);
}

int main() {
    UserId u = 5; //still CE
    UserId u2 = 5_uid;
}
```

Билет 4.7. Модификатор доступа protected для полей и методов. Приватное и публичное наследование. Разница между наследованием классов и структур. Разница между видимостью и доступностью. Видимость и доступность различных членов родителя в теле наследника при приватном и публичном наследовании, явное обращение к методам родителя, использование using.

protected — доступ открыт классам, производным от данного, и друзьям. То есть производные классы и друзья получают свободный доступ к таким данным или методам. Все другие классы такого доступа не имеют.

- Public-наследование.При таком наследовании protected и public данные из базового класса остаются, соответственно protected и public в производном классе. Все знают о факте наследования (имеют доступ к родителю через наследника).
- Private-наследование - означает, что поля, которые достались наследнику от родителей являются private, и к ним нельзя получить доступ извне, не будучи членом класса-наследника или другом класса-наследник. Derived запретил доступ к полям, наследованным из Base, поэтому даже если какая-то функция была другом Base, эта функция не имеет доступ к полям класса Derived, унаследованным из Base. Приватность устанавливается на уровне наследника.Кроме того, из наследника нельзя обращаться к приватной части родителя. О факте наследования знает только наследник.
- Protected-наследование. Данные, которые в Base были protected и public, становятся protected. О факте наследования знают только наследники и друзья.

  Поля класса могут быть protected.

Классы по умолчанию наследуются private, структуры — public.

Видимость — это то, какие функции и переменные видны в данной области видимости. Если несколько одноименных функций/переменных есть в области видимости, то, одна из них затмевает остальные, и, как следствие, остальные не видны в области видимости. Версия сына всегда затмевает версию родителя. Доступность — то, имеет ли данная область видимости доступ к перменной или функции. Золотое правило: Доступность проверяется после видимости и выбора перегрузки. Рассмотрим на примере:

```
1  class Base {
2  public:
3      int a = 0;
4      void f() {}
5      void g(int a) {}
6  };
7
8  class Derived : public Base {
9  protected:
10     int a = 1;
11 public:
```

```
12      void f() {}
13      void g() {}
14  };
15
16  int main() {
17      Derived d; //1
18      d.f(); // Вызовется та, что из Derived
19      d.g(3); // CE, g(int) доступна, но не видна
20      d.Base::f(); // OK
21      d.Base::g(3); // OK
22      d.a; // CE, видна, но недоступна
23      d.Base::a; //OK
24      return 0;
25  }
```

Добавить в область видимости g из Base можно, дописав в public часть класса Derived строчку using Base::g, аналогично при написании в private часть он был бы видим, но недоступен. При приватном наследование Base будет вне зоны видимости, поэтому даже сама запись d.Base будет уже CE.

Билет 4.8. Размещение объектов в памяти при наследовании. Порядок вызова конструкторов и деструкторов, а также инициализации полей при наследовании. Наследование конструкторов родителя с помощью using. Обращение к конструкторам родителя из конструкторов наследника. Множественное наследование. Проблема ромбовидного наследования, размещение объекта в памяти при таком наследовании.

Пусть класс Derived – наследник класса Base. Тогда в памяти объект типа Derived лежит так: [Base][Derived] (сначала все поля от родителя, а потом все поля от сына).

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:

```
1   class Base {
2   public:
3       int a;
4       Base(int a) : a(a) {};
5   };
6
7   class Derived : public Base {
8       int b;
9       //Derived(int b) : b(b)  нельзя писать, т.к. у Base нет конструктора по умолчанию
10      //Derived(int a, int b) : a(a), b(b)  тоже нельзя, потому что можно инициализировать только
        свои поля
11      Derived(int b) : Base(a), b(b) {};
12  };
```

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

Множественное наследование — наследование от нескольких классов.

Нужно быть аккуратными с полями классов – они могут быть одинаковы у обоих родителей.

```
class Mother {
public:
    int a = 1;
};
class Father {
public:
    int a = 2;
};
class Son: public Mother, public Father {
    int s = 3;
};
int main() {
    Son s;
    cout << s.a;
    //CE: request for member "a"is ambiguous
}
```

Проблема ромбовидного наследования Рассмотрим следующий код, который при $Granny\&g = s$; выдаст неоднозначный каст. Тут две разные бабушки лежат в M и F, если обратимся к полю g у сына, будет CE. Размер сына 20, там две копии g. Сын в памяти лежит как [g][m][g][f][s]

```
struct Granny {
    int g = 0;
};
struct Mother: public Granny {
    int m = 1;
};
struct Father: public Granny {
    int f = 1;
};
struct Son: public Mother, public Father {
    int s = 3;
};
```

Еще один пример:

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

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

int main()
```

```
{
    C c;
    c.a; //CE
    c.f(); //тоже нельзя
    c.B1::f(); // можно
}
```

Проблема снова 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. A в выражении (3) всё хорошо, потому что однозначно. (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”.

Поскольку в C++ при инициализации объекта дочернего класса вызываются конструкторы всех родительских классов, возникает и другая проблема: конструктор базового класса бабушки будет вызван дважды.

Про **using**:

```
struct B {
    B(int = 13, int = 42);
};
struct D : B {
    using B::B;
};
```

Еще примеры.

```
struct B1 {  B1(int, ...) { } };
struct B2 {  B2(double)   { } };

int get();

struct D1 : B1 {
  using B1::B1;   // inherits B1(int, ...)
  int x;
  int y = get();
};

void test() {
  D1 d(2, 3, 4); // OK: B1 is initialized by calling B1(2, 3, 4),
                 // then d.x is default-initialized (no initialization is performed),
                 // then d.y is initialized by calling get()
  D1 e;          // Error: D1 has no default constructor
}

struct D2 : B2 {
  using B2::B2; // inherits B2(double)
  B1 b;
};
```

```
23
24 D2 f(1.0);          // error: B1 has no default constructor
```

Как и в случае использования-объявлений для любых других нестатических функций-членов, если унаследованный конструктор соответствует сигнатуре одного из конструкторов Derived, он скрывается от поиска версией, найденной в Derived. Если один из унаследованных конструкторов Base имеет сигнатуру, которая соответствует конструктору копирования / перемещения производного, это не предотвращает неявную генерацию производного конструктора копирования / перемещения (который затем скрывает унаследованную версию).

```
1 struct B1 {   B1(int); };
2 struct B2 {   B2(int); };
3
4 struct D2 : B1, B2 {
5   using B1::B1;
6   using B2::B2;
7   D2(int);    // OK: D2::D2(int) hides both B1::B1(int) and B2::B2(int)
8 };
9 D2 d2(0);    // calls D2::D2(int)
```

4.9. Шаблонные функции, синтаксис их объявления. Перегрузка и специализация шаблонных функций, разница между этими понятиями (на примере). Эвристические правила разрешения перегрузки: «частное лучше общего», «точное соответствие лучше приведения типа». Примеры разрешения перегрузки между шаблонными функциями.

1. **Синтаксис объявления шаблонных функций:**

```
template <typename T>
T max(const T& a, const T& b) {
    return a > b ? a : b;
}
```

2. **Перегрузка:**

```
template <typename T, typename U>
void f(T, U) { std::cout << 1; }

template <typename T>
void f(T, T) { std::cout << 2; }

void f(int, double) { std::cout << 3; }
```

   Рассмотрим несколько вызовов f и посмотрим что выведется

   - `f(0, 0)` - выведется 2, так как он «более частная» чем 1, то есть подходит в меньшем количестве случаев + в ней не требуется приведение типов как в 3
   - `f(0, 0.0)` - выведется 3, так как она более частная чем 1, а 2 просто не подходит, так как непонятно, чему равно T
   - `f(0.0, 0)` - выведется 1, так как в 3 нужно сделать 2 приведения типа, а это хуже чем точное совпадение.

3. **Специализация:** по стандарту в функциях запрещена частичная специализация, так как этот механизм уже реализован в виде перегрузки.

```
template <typename T, typename U>
void f(T, U) { std::cout << 1; }

template <typename T>
void f(T, T) { std::cout << 2; }

template <>
void f(int, int) { std::cout << 3; }
```

   Важно запомнить, что «хозяйкой» специализации является функция, объявленная над ней. Рассмотрим вызов `f(0, 0)`. Очевидно, что в перегрузке выберется вторая версия, а затем выберется специализация и в итоге выведется 3.

   Поменяем в коде функции 2 и 3 местами. Теперь хозяйкой специализации стала первая функция. В перегрузке все так же побеждает вторая, но специализации у нее больше нет, поэтому выведется 2.

   Итог: Специализация не участвует в перегрузке, рассматривается только если ее хозяйка победила

4.10. Специализация шаблонов. Синтаксис объявления специализации для функций и классов. Разница между перегрузкой и специализацией шаблонных функций на примере. Частичная и полная специализация шаблонов классов. Примеры: реализация hash для нестандартного типа, реализация is_same, реализация remove_reference.

1. **Специализации классов:**

```cpp
template <typename T>
struct vector {};

template <typename T> // частичная специализация
struct vector<T*> {}; // можно T&, const T, T[], etc.

template <> // полная явная специализация
struct vector<bool> {};
```

2. **Реализация hash:** (нигде не было лишь мои догадки. после консультации уточню)

```cpp
struct A { std::string field; }
template <typename T>
struct hash {};
template <>
struct hash<A> {
    size_t operator () (const A& a) {
        return std::hash<std::string>(a.field); // как-то хэшируем a
    }
};
```

3. **Реализация is_same:**

```cpp
template <typename U, typename V>
struct is_same {
    static const bool value = false;
};

template <typename U>
struct is_same<U, U> {
    static const bool value = true;
};
```

4. **Реализация remove_reference:**

```cpp
template <typename T>
struct remove_reference {
    using type = T;
};

template <typename T>
struct remove_reference<T&> {
    using type = T;
};

template <typename T>
struct remove_reference<T&&> {
    using type = T;
};
```

4.11. Простейшие compile-time вычисления с помощью шаблонной рекурсии. Вычисление чисел Фибоначчи в compile-time с помощью шаблонной рекурсии. Проверка на простоту числа N за O(N) в compile-time с помощью шаблонной рекурсии.

Вычисление N-ого числа Фибоначчи

```
template <size_t N>
struct Fibonacci {
  static const size_t value = Fibonacci<N-1>::value + Fibonacci<N-2>::value;
};

template <>                                  template <>
struct Fibonacci<1> {                        struct Fibonacci<0> {
    static const size_t value = 1;               static const size_t value = 0;
};                                           };
```

Проверка числа N на простоту за $O(N)$

```
template <size_t N, size_t D>
struct IsPrimeHelper { // проверка что N не делится на все числа ≤ D
    static const bool value = (N % D == 0 ? false
                                          : IsPrimeHelper<N, D-1>::value);
};

template <size_t N>
struct IsPrimeHelper<N, 1> { // база
    static const bool value = true;
};

template <size_t N>
struct IsPrime {
    static const bool value = IsPrimeHelper<N, N-1>::value;
};

template <>
struct IsPrime<1> { // отдельно случай для 1
    static const bool value = false;
};
```

Замечание: Максимальная глубина шаблонной рекурси по дефолту равна 1024. Ее можно увеличить с помощью флага `-ftemplate-depth=`новая глубина (но если сильно увеличить могут вылезти другие страшные ошибки)

4.12. Виртуальные функции. Объяснение логики выбора версии метода у наследника в случаях, когда функции виртуальные и когда нет. Логика выбора версий в случае, когда присутствуют как виртуальные, так и невиртуальные методы со слегка отличающимися типами параметров, разной константностью, разной приватностью и т. п.. Логика выбора версий в случае многоуровневого наследования и в случае множественного наследования.

Полиморфизм – один из главных столпов объектно-ориентированного программирования. Его суть заключается в том, что один фрагмент кода может работать с разными типами данных. (Например: операция плюс для целых чисел, для рациональных чисел, для матриц. Операция называется одинаково, но в зависимости от типа выполняются разные действия)

Виртуальная функция – это такая функция, что если к наследнику обратиться через ссылку на родителя, то выберется версия наследника.

```
#include <iostream>

struct Base {
    virtual void f() {std::cout << 1;}
    virtual void h() {std::cout << 1;}
    void g() {std::cout << 1;}
}

struct Derived: public Base {
    void f() {std::cout << 2;}
    virtual void h() {std::cout << 2;}
    void g() {std::cout << 2;}
}

struct Subderived: public Derived {
    void f() {std::cout << 3;}
}

int main() {
    Derived d;
    Base& b = d;
    b.f() // output: 2
    b.h() // output: 2
    b.g() // output: 1

    Subderived s;
    Base& b2 = s;
    b2.f() // output: 3 //Логика выбора версий в случае многоуровневого
}
```

При виртуальном наследовании при вызове метода вызывается более частная версия метода.

Выбор версии между виртуальной и невиртуальной

```
    struct Base {
        virtual void f() { cout << 1; }
    };
```

```
    struct Derived: public Base {
        void f() const { cout << 2; }//(1)
        virtual void f() const { cout << 2; } // (2)
    }

    int main() {
        Derived d;
        Base& b = d;
        b.f() // output: 1
    }
```

(1) - не виртуальный! потому что не полностью совпадает по сигнатуре. Т.е. небольшое изменение сигнатуры виртуальной функции приводит к тому, что она перестаёт быть виртуальной.

(2) - это всё ещё второй метод f, но уже виртуальный т.е. всё ещё не переопределяет Base f

Для виртуальных функций выбор происходит в Runtime, для не виртуальных в Compile time

Логика выбора версий в случае, когда присутствуют как виртуальные, так и невиртуальные методы со слегка отличающимися типами параметров, разной константностью, разной приватностью и т.п.:

Компилятор (в Compile time) для вызова выбирает версию функции, не смотря на виртуальность, а смотря на другие критерии (список аргументов, константность и т.д.). Поэтому среди виртуальных и невиртуальных функция выбирается функция вне зависимости от виртуальности. Если была выбрана виртуальная функция то в Runtime выбирается нужная версия этой виртуальной функции.

Логике множественного наследования.

Просто пример, лектор на эту тему особо не говорил.

```
struct Base1 {
    virtual void f() { cout << 1; }
}

struct Base2 {
    virtual void f() { cout << 2; }
}

struct Derived: public Base1, public Base2 {
    void f() {cout << 3;}
}

int main() {
    Derived d;
    Base1& b1 = d;
    b1.f() // output: 3
}
```

4.13. Чисто виртуальные функции, синтаксис определения, примеры использования. Понятие абстрактных классов. Виртуальный деструктор, особенности его определения и пример проблемы, возникающей в случае его отсутствия. Понятие RTTI, оператор typeid и особенности его использования.

```
class AbstractAnimal() {
    virtual int getAge(); // обычная виртуальная функция
    virtual void make_sound() = 0; // чисто виртуальная функция
}
```

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, мы просто присваиваем ей значение 0.
Использование чистой виртуальной функции имеет два основных последствия. Во-первых, любой класс с одной и более чистыми виртуальными функциями становится абстрактным классом, объекты которого создавать нельзя.
Во-вторых, все дочерние классы абстрактного родительского класса должны переопределять все чистые виртуальные функции, в противном случае — они также будут считаться абстрактными классами.

Проблема виртуального деструктора (при отсутствии виртуальности).

```
struct Base {
    int* x = new int();
    ~Base() {
        delete x;
    }
}

struct Derived: public Base {
    int* y = new int();
    ~Derived() {
        delete y;
    }
}

int main() {
    Base* b = new Derived();
    delete b; // здесь вызовется деструктор для Base, а поле Derived::y не будет разрушено - утечка памяти
}
```

Проблема решается объявлением деструктора виртуальным методом.

RTTI (Run-time type information)

Динамическая идентификация типа данных (RTTI) — механизм, который позволяет определить тип данных переменной или объекта во время выполнения программы.

Если тип является полиморфным, то в compile time нельзя однозначно узнать, какую версию функции надо выбрать. Пример:

```
#include <iostream>
```

```
2
3  struct Base {
4      virtual void f() {
5          std::cout << 1;
6      }
7      virtual ~Base() = default;
8  }
9
10 struct Derived: public Base {
11     void f() override { //override - некоторое обязательство того, что функция
       виртуальная
12         std::cout << 2;
13     }
14 }
15
16 int main() {
17     int x;
18     std::cin >> x;
19     Base b;
20     Derived d;
21     Base& bb = x > 0 ? b : d;
22     bb.f();
23 }
```

int main() компилируется, если типы кастуются, что тут и происходит

Встроенный оператор typeid() - позволяет узнать тип выражения, известный компилятору (в Rantime). Реальный тип может быть другим, значит, компилятор вынужден в Runtime поддерживать информацию о типе. По этой же причине нельзя отследить проблему приватности и выбора версии.

```
1  struct Base {
2  }
3
4  struct Derived: public Base {
5  }
6
7  int main() {
8      Base b;
9      Derived& d1;
10     Derived& d2 = b;
11     cout << (typeid(b) == typeid(d1)) << endl; //false
12     cout << (typeid(b) == typeid(d2)) << endl; //true
13     cout << typeid(b).name() << endl; //3Base - некоторое строковое название
       структуры
14 }
```

4.14 Приведения типов при наследовании: static_cast, dynamic_cast и reinterpret_cast. Особенности использования каждого из этих операторов. Пример ситуации, когда все три этих оператора ведут себя по-разному. Способы приведения “вверх”, “вниз”, и “вбок” по иерархии наследования. Особенности использования данных операторов при приватном наследовании, множественном наследовании.

- **static_cast**: принимает решение на этапе компиляции и может, в частности:
  - Вызвать конструктор или определённый пользователем оператор преобразования типа — в частности, помеченный как explicit
  - Преобразовать тип указателя или ссылки в случае наследования и ряде других
  - Использовать стандартное преобразование типа. 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)

```
Base b;
Derived d;

static_cast<Base>(d);
//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
```

static_cast позволяет кастовать и вниз, но будет UB
static_cast проверяет легальность каста. То есть если мы пытаемся кастовать вверх к классу, от которого наследовались приватно, нам не дадут этого сделать :(

Не сработает

Так тоже

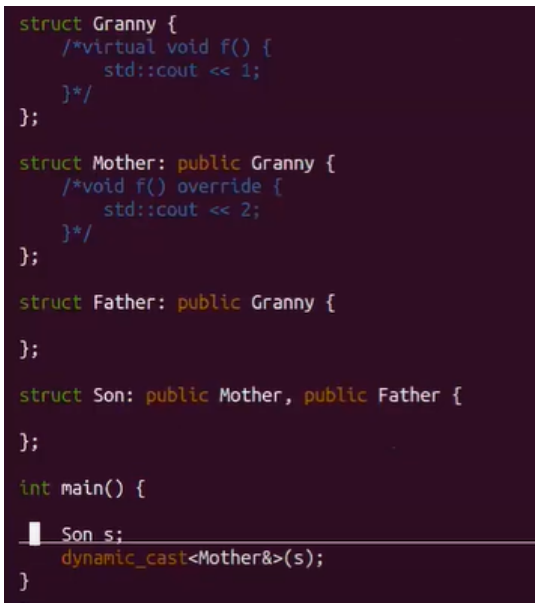
- **reinterpret_cast**: более топорно меняет тип выражения. Не выполняет никаких дополнительных операций в рантайме. Разрешаются любые преобразования указателей, не понижающие константность. Благодаря этому, в отличие от static_cast, можно преобразовывать указатель на наследника к родителю при private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

- **dynamic_cast** dymamic_cast это такое преобразование типов, которое в runtime проверяет, действительно ли тип того, на что мы сейчас указываем, совместим с типом того, к чему мы хоим скастовать. Если да, то выполняется преобразование типов, иначе - RE (можно навесить исключение).

  dynamic_cast можно делать как к ссылке, так и к указателю

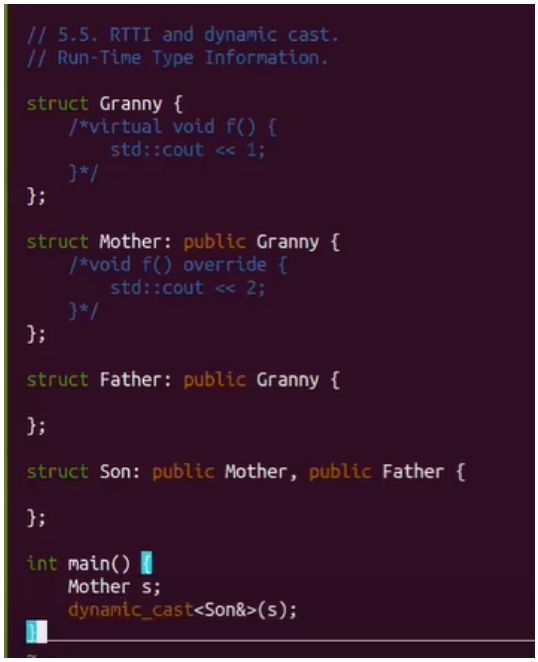
  dymamic_cast может:

  ✓ Делать каст вверх и не для полиморфных типов



Сработает

× **Не может** делать каст вниз **не для полиморфных** типов, потому что он не может проверить в runtime, возможен ли каст, ибо типы не полиморфные



Не сработает

× Если тип полиморфный и под указателем лежит родительский тип, то каст вниз сработает, но будет неуспешным (если каст прошел неуспешно, то dynamic_cast<...*>(...) выдаст nullptr, а dynamic_cast<...>(...) - ошибку std::bad_cast)

Скомпилируется, но не скастуется

Так произошло потому, что изначально под указателем лежал не сын, а мама. Но если бы мы изначально указатель на маму проинициализировали сыном, все бы сработало

✓ (То же самое, но по указателю на маму лежит сын)

```
struct Granny {
    virtual void f() {
        std::cout << 1;
    }
};

struct Mother: public Granny {
    /*void f() override {
        std::cout << 2;
    }*/
};

struct Father: public Granny {

};

struct Son: public Mother, public Father {

};

int main() {
    Mother* pm = new Son();

    std::cout << dynamic_cast<Son*>(pm);

    delete pm;
}
```

Сработает

✓ И вот так dynamic_cast тоже со всем справится

```
struct Granny {
    virtual void f() {
        std::cout << 1;
    }
};

struct Mother: public Granny {
    /*void f() override {
        std::cout << 2;
    }*/
};

struct Father: public Granny {

};

struct Son: public Mother, public Father {

};

int main() {
    Mother* pm = new Son();

    std::cout << dynamic_cast<Father*>(pm);

    delete pm;
}
```

Сработает

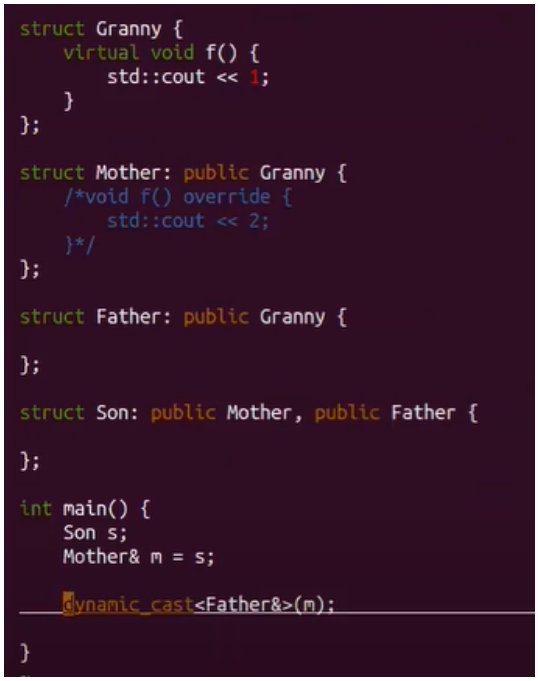
Что в этом случае происходит? dynamic_cast смотрит, возможно ли теоретически прикастовать маму к папе. Да! Это возможно, так как они лежат в одном графе, а значит, под указателем может лежать сын, и чисто теоретически мы могли бы прикастоваться. Но проверить, кто там лежит, мы можем лишь в runtime.

* Если бы под указателем на маму лежала мама, то в runtime мы бы это засекли и dynamic_cast вернул бы nullptr
* Если под указателем на маму лежит сын, то каст корректен, и все нормально кастуется, куда надо

Пример ситуации, когдавсе три этих оператора ведут себя по-разному
Собственно, последний пример работы dynamic_cast нам подходит

* **dynamic_cast** корректно отработает и прикастует маму к папе
* **static_cast** не сработает, так как static_cast работает только вверх-вниз, но не вбок (CE)
* **reinterpret_cast** выдаст UB (так как мы пытаемся кастовать несовместимые типы)

Тут мы можем кастовать не только к указателю, но и к ссылке. dynamic_cast по-прежнему молодец, reinterpret_cast все еще выдает UB, а static_cast не работает



Сработает

4.15 Исключения. Преимущества и недостатки использования исключений. Особенности копирования исключений при их создании и при поимке. Разница между throw без параметров и throw с параметром. Правила приведения типов при поимке исключений. Правила выбора подходящей секции catch

Преимущества и недостатки

- ✓ Исключения отделяют код обработки ошибок от нормального алгоритма программы, тем самым повышая разборчивость, надежность и расширяемость кода.
- ✓ Исключения легко передаются из глубоко вложенных функций.
- ✓ Генерация исключения – единственный чистый способ сообщить об ошибке из конструктора
- ✓ Исключения могут быть, и часто являются, определяемыми пользователем типами, несущими гораздо больше информации, чем код ошибки.
- × Исключения нарушают структуру кода, создавая множество скрытых точек выхода, что затрудняет чтение и изучение кода.
- × Исключения тяжело ввести в устаревший код.
- × Исключения неверно используются для выполнения задач, относящихся к нормальному алгоритму программы.

Особенности копирования исключений приих создании и при поимке

Рассмотрим вот такой код:

```
struct Noisy {
    Noisy() {
        std::cout << "created";
    }
    Noisy(const Noisy&) {
        std::cout << "copy";
    }
    ~Noisy() {
        std::cout << "destroyed";
    }
};

void f() {
    Noisy x;

    throw x;
}


int main() {
    try {
        f();
    } catch (const Noisy& x) {
        std::cout << "caught";
    }
}
```

Что тут произойдет при запуске? Обратим внимание на функцию f. В ней создается локальный объект x, который нам предстоит бросить. Но при выходе из функции все локальные объекты уничтожаются, поэтому мы бросаем не x, а делаем копию x, а потом уже бросаем ее. Таким

образом, программа выведет: "created copy destroyed caught destroyed"
Если мы бы принимали не по константной ссылке, а просто по значению, создалась бы еще одна копия после того, как мы скопировали объект для того чтобы бросить

А без конструктора копирования не получится бросить :(

Разница между throw без параметров и throw с параметром.

throw с параметром используется для того, чтобы бросить какой-то конкретный объект, а throw без параметров используется только внутри catch, если необходимо пробросить уже пойманное исключение наверх.
Существенное отличие заключается в том, что **throw без параметров не создает копию**
Рассмотрим пример

```
void g() {
    try {
        throw std::out_of_range("aaa");
    } catch (std::exception& ex) {
        std::cout << "caught";
        throw;
    }
}

int main() {
    try {
        g();
    } catch (std::out_of_range& oor) {

    }
```

Сейчас исключение поймается нормально, потому что в main прилетело std::out_of_range

Но если заметить throw на throw ex, то мы уже в main его не поймаем.

```
void g() {
    try {
        throw std::out_of_range("aaa");
    } catch (std::exception& ex) {
        std::cout << "caught";
        throw ex;
    }
}

int main() {
    try {
        g();
    } catch (std::out_of_range& oor) {
        std::cout << "caught2";
    }
```

std::out_of_range уничтожилось, и дальше полетело то, что мы поймали, но поймали мы родителя. Тем самым мы создали копию родителя, и дальше полетел родитель, а не std::out_of_range

Правила при поимке исключений

1. В catch не работает приведение типов за исключением ситуаций, когда это родитель и наследник. Поймать наследника по ссылке или копии на родителя можно. А вот такой код не поймает ошибку:

```
int main() {
    try {
        throw 1;
    } catch (unsigned int x) {

    }
```

2. Работает приведение типов между const и не const
3. Если у нас есть несколько catch, то компилятор выбирает первый подходящий

```
try {
    throw std::out_of_range("aaaa");
} catch (std::exception& ex) {
    std::cout << 1;
} catch (std::out_of_range& oor) {
    std::cout << 2;
} catch (...) {
    std::cout << 3;
}
```

Здесь выведется 1

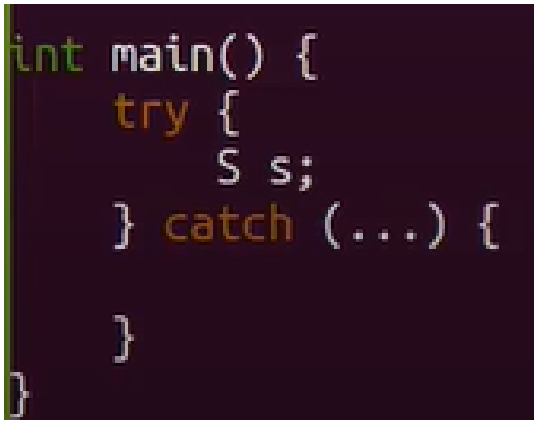
4.16 Проблема исключений в конструкторах. Поведение программы при выбросе исключения из конструктора. Идиома RAII. Проблема исключений в деструкторах. Функция uncaught_exception, ее предназначение. Спецификации исключений: спецификатор noexcept и оператор noexcept, синтаксис и пример применения. Исключения в списках инициализации конструкторов, function-try блоки

Проблема исключений в конструкторах.Поведение программы при выбросе исключения из конструктора. Идиома RAII.

Пусть у нас есть вот такой класс

```
struct S {
    int* p = nullptr;

    S(): p(new int (5))
        throw 1;
    }

    ~S() {
        delete p;
    }
};
```

Хотим выполнить такой код



Произойдет утечка памяти. Вопрос. Должен ли вызваться деструктор, если исключение вылетело из конструктора? Если исключение вылетает из конструктора, объект еще не до конца создан, и значит, комплятор не может вызать деструктор для него. Как решать такую проблему?

Идиома **RAII - Resourse Acquisition Is Inizialization** - захват ресурса есть инициализация некоторого объекта. Всякий раз, когда нужно захватить какой-то ресурс, это делаем не в лоб, а с помощью объекта, который явно это делает. Тогда каждый раз когда будет выбрасываться исключение, деструкторы локальных объектов гарантированно вызовутся и, следовательно, ресурсы будут освобождены вовремя.

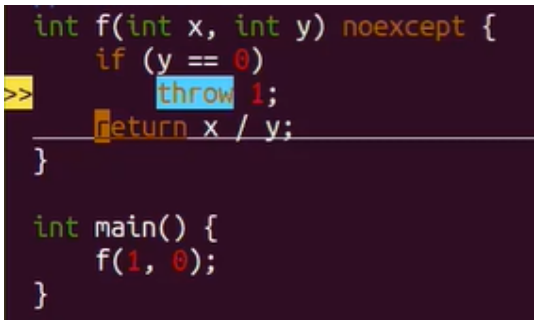
```
template <typename T>
class SmartPtr {
private:
    T* ptr;
public:
    SmartPtr(T* ptr): ptr(ptr) {}
    ~SmartPtr() {
        delete ptr;
    }
};


struct S {
    SmartPtr<int> p = nullptr;

    S(): p(new int (5)) {
        throw 1;
    }

    ~S() {}
};


void f() {
    SmartPtr<int> p = new int(5);

    throw 1;
}
```

Мы обернули указаль в класс SmartPtr, который создается в теле функции. Поэтому, будучи локальным объектом, он удалится в случае выхода из функции и утечки памяти не произойдет

Спецификации исключений: спецификатор noexcept и оператор noexcept, синтаксис и пример применения.

Подобно тому, как мы пишем, что функция может работать с константными объектами (используя ключевое слово const), можно также помечать, безопасна ли функция с точки зрения исключений с помощью слова "noexcept".

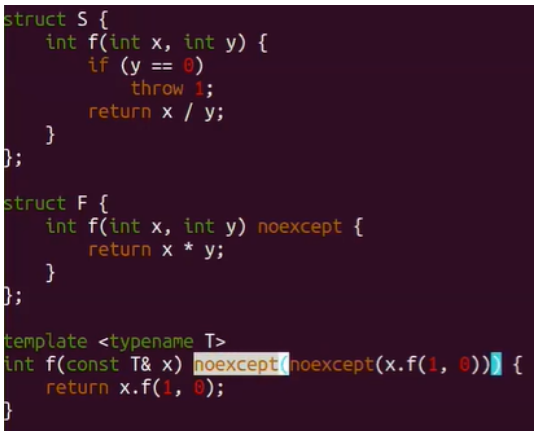
Если все-таки запихнуть throw в noexcept функцию, будет RE



Спецификатор noexcept

Теперь посмотрим вот на такой код(картинка ниже). Здесь есть функция f, которая бывает то не noexcept, то noexcept в зависимости от шаблонного параметра. По хорошему, эту функцию нужно бы либо пометить, либо не пометить noexcept в зависимости от того, является ли noexcept вызов внутри функции f. Для этого существует условный noexcept.

На картинке ниже выделенный noexcept и невыделенный рядом с ним - это разные по смыслу noexcept. Выделенный является ключевым словом, невыделенный - оператором.

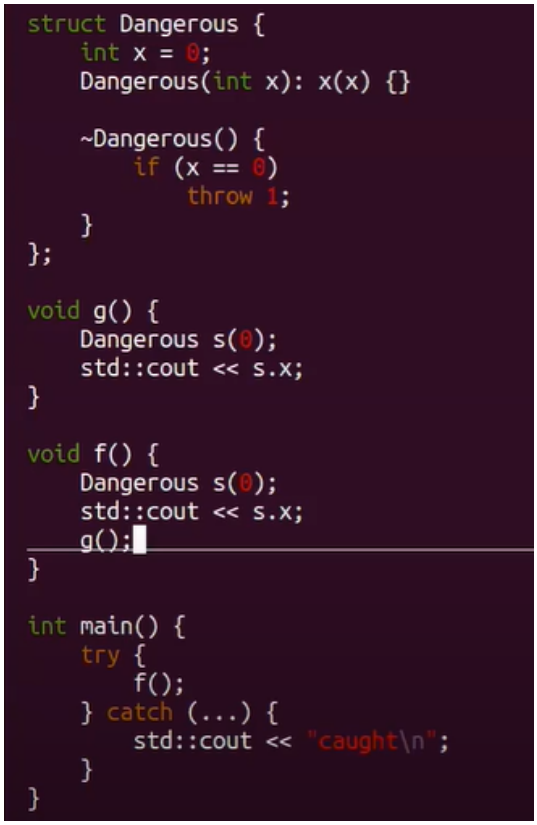


Оператор noexcept(...) возвращает true, если выражение в скобках безопасно относительно исключений и false в ином случае.

Замечание Аналогично оператору sizeof noexcept не вычисляет значение внутри него, а просто осуществляет проверку на наличие выражений, которые потенциально могут бросить исключение. Среди них:

- throw
- new
- dynamic_cast
- noexcept function call

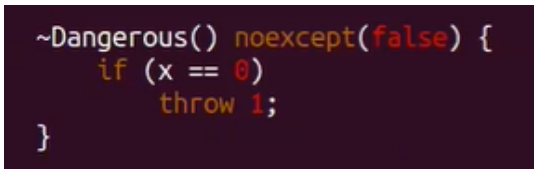
Проблема исключений в деструкторах.



Давайте проанализируем, что будет, если мы запустим этот код. Вот мы вызвали f, она создала опасный объект s, вызвала g, которая создала опасный объект s. При завершении g уничтожаются все локальные объекты, в том числе и s. Деструктор Dangerous кидает исключение, которое завершает работу g, и мы вылетаем в f вместе с этим исключением. f тоже должна завершиться, удаляются все локальные объекты, мы переходим в деструктор Dangerous... и ловим еще одно исключение! В момент, когда исключение уже летело. И программа падает
Что в этом случае делать?

Не бросать исключения из деструкторов

По умолчанию, начиная с C++11 все деструкторы считаются noexcept функциями
Если очень надо бросить исключение из деструктора, можно сделать так (но не нужно):



Исключения в списках инициализации конструкторов, function-try блоки

Пусть у нас есть какой-то класс, у которого мы инициализируем поля списком инициализации.
И внезапно один из вызовов в списке инициализации бросает исключение

```
class A {
public:
  A(int n) {
    throw 0;   // Конструктор класса A бросает исключение int
  }
};

class B {
public:
  B(int n);
private:
  A __a;
};

B::B(int n)
  : __a(n)     // Данный вызов бросает исключение
{}
```

Для решения этой проблемы используют try-catch блоки

Function-try block

Когда нам нужно обернуть всё тело функции в try, то можно использовать спецификатор **try**:

```
1     void f() try{
2     } catch() {}
```

Тогда наш код со списком инициализации переписывается следующим образом:

```
class B {
public:
  B();
private:
  P* __p;
  A __a;
};

B::B()
try
  : __p(new P), __a(0) {
} catch (int& e) {
  std::cout << "B(), exception " << e << std::endl;
  delete __p;
};
```

Функция uncaught_exception, ее предназначение.

https://en.cppreference.com/w/cpp/error/uncaught_exception

1 Определяет, есть ли у текущего потока активный объект исключения. Было исключение брошено, брошено лететь дальше или еще не поймано, std::terminate or std::unexpected. Другими словами, std :: uncaught _exception определяет, выполняется ли в настоящий момент раскрутка стека.

2 Отслеживает, сколько ошибок было брошено/летит в текущем потоке или еще не поймано

4.17 Понятие аллокатора. Зачем нужны аллокаторы? Реализация основных методов std::allocator. Оператор placement new, его синтаксис использования и отличие от обычного new. Структура std::allocator_traits, ее предназначение. Структура rebind, ее предназначение.

Оператор placement new, его синтаксис использования и отличие от обычного new

Напоминание: Если выделен какой-то кусок памяти под S, но конструктор на нем еще не было вызван, то есть синтаксис чтобы направить конснhуктор на уже выделенную память по указателю

```
    S* p = reinterpret_cast<S*>(operator new(sizeof(S))); // сырая память
    S* p1 = reinterpret_cast<S*>(operator new(sizeof(S)));// сырая память
    new(p) S(); //default construct will be called
    new (p1) S(value); //construct from value



    void* operator new(size_t , S* p){ // new for placement new
        return p;
    }
```

Замечание: если в структуре переопределен оператор new, то placement new для него сам по себе не генерируется

В чем отличие?

Обычный new выделает память, а потом вызывает на этой памяти конструктор, а placement new не выделяет память, а просто вызывает конструктор
Замечание: placement delete не существует

Понятие аллокатора. Зачем нужны аллокаторы? Реализация основных методов std::allocator

Оператор new - довольно низкоуровневая абстракция, поэтому чтобы работать на более высоком уровне, на уровне языка программы, а не на уровне операционной системы, придумали выделять и владеть памятью в виде класса. Аллокатор "стоит" между контейнером и оператором new.

Стандартный аллокатор

```
    template<typename  T>
    struct allocator{
        T* allocate(size_t n) {
            return::operator_new(n * sizeof(T));
        }

        void deallocate(T* ptr, size_t n) {
            ::operator delete(ptr);
        }


        template<typename... Args>
        void construct(T* ptr, const Args&... args){
            new(ptr) T(args...);
        }
```

```
        void destroy(T* ptr){
            ptr->~T();
        }
    }
```

allocate - выделяет нужное число байт
deallocate - очищает память по указателю
construct - конструирует объект(вызывает конструктор) на том месте, куда указывает указатель(просто вызывается placement new)
destroy - вызывает деструктор у объекта, лежащего по указателю

Структура std::allocator_traits, ее предназначение

std::allocator_traits создан для того, чтобы некоторые вещи доопределить за аллокатор, так как некоторые методы в практически всех аллокаторах делают одно и то же. Например с методом construct: если в аллокаторе он определен, то вызывается он, иначе вызовется метод, определенный в allocator_traits. Структура со статическими методами - ссылка на аллокатор и то что нужно передать аллокатору (из-за того что там только методы, то нельзя создать объект этого класса)

Структура rebind, ее предназначение.

Если тип того что надо выделять на алокаторе совпадает с типом шаблонного параметра, то проблем нет. Однако в таком контейнере как лист, это не выполняется. С C++17 определен в allocator_traits. По сути, подменяет один шаблонный параметр на другой

```
template<typename T, typename Alloc = std::allocator<T>>
class list{
    class Node{};
    typename std::allocator_traits<Alloc>::template rebind_alloc<Node> alloc;
public:
    list(const Alloc& alloc = Alloc()) : alloc(alloc) {};
};

// realization
template< class U >
struct rebind {
    typedef allocator<U> other;
};
```

4.18 Понятие итератора. Пример реализации итераторов для любого из стандартных контейнеров (на ваш выбор). Разница между константными и неконстантными итераторами. Реализация константных итераторов без дублирования кода относительно обычных итераторов, применение метафункции std::conditional.

Разница между константными и неконстантными итераторами заключается в том, что элементы, на которые указывает константный итератор, нельзя модифицировать При разыменовании const итератора мы получаем не ссылку на T, а const ссылку на T.

Можно инициализировать константный итератор неконстантным итератором, но не наоборот

Пример реализации для List

```
template <bool IsConst>
   class myiterator {
   public:
       Node* iter = NULL;
       using difference_type = std::ptrdiff_t;
       using value_type = std::conditional_t<IsConst, const T, T>;
       using pointer = std::conditional_t<IsConst, const T*, T*>;
       using reference = std::conditional_t<IsConst, const T&, T&>;
       using iterator_category = std::bidirectional_iterator_tag;


       myiterator() {}
       myiterator(Node* iter) : iter(iter) {}
       reference operator*() {
           return (iter->data);
       }

       pointer operator->() { return &(iter->data); }

       myiterator& operator ++() {
           Node* ptr = iter->next;
           iter = ptr;
           return *this;
       }

       myiterator operator ++(int) {
           myiterator copy = *this;
           Node* ptr = iter->next;
           iter = ptr;
           return copy;
       }

       myiterator& operator --() {
           iter = iter->prev;
           return *this;
       }

       bool operator !=(const myiterator& other) {
           return !(iter == other.iter);
       }
       bool operator ==(const myiterator& other) {
           return (iter == other.iter);
       }
```

```
44          operator myiterator<true>() { return myiterator<true>{iter}; }
45      };
46
47      using const_iterator = myiterator<true>;
48      using iterator = myiterator<false>;
```

Реализация std::conditional

```
template<bool condition, typename T, typename F>
struct conditional{
    using type = F;
}
template<bool condition,typename T, typename F>
struct conditional<true, F, T>{
    using type = T;
}
template<bool condition, typename T, typename F>
using conditional_t = typename conditional<condition,T,F>::type;
```

4.19 Функции std::advance и std::distance, пример их применения. Реализация функции advance с правильной поддержкой разных видов итераторов, два способа такой реализации: с помощью перегрузки функций (старый способ до C++17) и с помощью if constexpr (новый способ, начиная с C++17)

В стандартной библиотеке есть функции std::advance и std::distance, которые позволяют сделать следующее:

- Функция std::advance берет итератор и продвигает его на указанное количество шагов.

```
std::list<int> v = {1, 2, 3, 4, 5};

std::list<int>::iterator it = v.begin();
std::advance(it, 3);
std::cout << *it;
```

Выведет 4

- Функция std::distance берет два итератора и считает расстояние между ними (сколько шагов нужно пройти, чтобы пройти от одного итератора до другого)

```
std::list<int>::iterator it2 = v.end();
std::cout << std::distance(it, it2);
```

Выведет 2

```
std::list<int>::iterator it2 = v.end();
std::cout << std::distance(it2, it);
```

А вот так UB

Как реализовать advance? Ведь если мы передаем ей forward/bidirectional итераторы, то она сможет отработать только за линейное время прибавлением по единичке, но если мы передаем ей random access iterator, то такое прибавление можно сделать за O(1).

Возникает проблема, как понять, какой именно итератор передан в функцию на вход, чтобы сделать реализацию наиболее эффективной.

В стандартной библиотеке есть такая структура std::iterator_traits

https://en.cppreference.com/w/cpp/iterator/iterator_traits

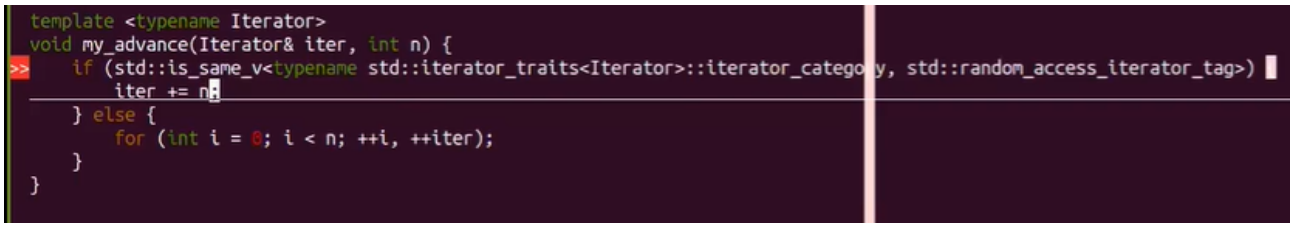
Среди прочего, в нем определены

Member types

Member type	Definition
difference_type	Iter::difference_type
value_type	Iter::value_type
pointer	Iter::pointer
reference	Iter::reference
iterator_category	Iter::iterator_category

- **value_type** - тот тип, на который ссылается итератор
- **pointer** - C-style указатель на объект под итератором
- **reference** - ссылка на объект под итератором
- **iterator_category** - это некоторый typedef, который позволяет понять, какой категории итератор нам дан

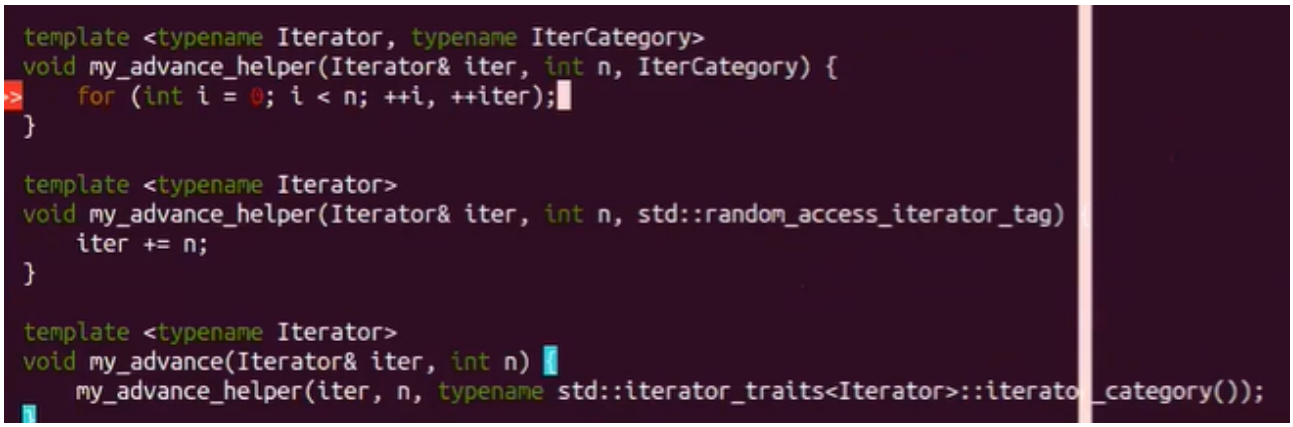
Попробуем реаллизовать нашу функцию



Это не сработает!

В чем, собственно, проблема? Проблема в том, что мы логически понимаем, что в ветку if мы не войдем, а компилятор не может это проверить на этапе компиляции и считает, что мы пытаемся сделать += к итератору, для которого данная функция не определена

Как нужно было делать (до C++17)



Как нужно делать (с C++17)

```
template <typename Iterator>
void my_advance (Iterator& it, int n) {
    if constexp ( std::is_same_v<typename std::iterator_traits<Iterator>::
   iterator_category, std::random_access_iterator_tag>) {
        it += n;
    } else {
        for (int i = 0; i < n; ++i; ++it);
    }
}
```

constexp означает *не компилировать одну из веток if, если условие ложно*, но условие должно проверяться в compile time

Extra

reverse_iterator

reverse_iterator - это такой итератор, который делает то же самое, что и обычный итератор, но в порядке не слева направо, а справо налево

```
using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

     reverse_iterator rbegin() {
        return reverse_iterator(end());
    }
    reverse_iterator rend() {
        return reverse_iterator(begin());
    }

    const_reverse_iterator rbegin() const {
        return const_reverse_iterator(end());
    }
    const_reverse_iterator rend() const {
        return const_reverse_iterator(begin());
    }

    const_reverse_iterator crbegin() const {
        return const_reverse_iterator(end());
    }
    const_reverse_iterator crend() const {
        return const_reverse_iterator(begin());
    }
```

У reverse_iterator есть функция base(), возвращающая нормальный итератор, от которого взят reverse_iterator

back_inserter

```
template <typename Container>
class back_insert_iterator{
    Container& container;
public:
    back_insert_iterator(Container& container): container(container){}

    back_insert_iterator<Container>& operator++(){
        return *this;
    }

    back_insert_iterator<Container>& operator*(){
        return *this;
    }
    back_insert_iterator<Container>& operator=(const typename Container::
   value_type& value){
        container.puch_back(value);
        return *this;
    }
}

template <typename Container>
back_insert_iterator<Container> back_inserter(Container& container){
    return back_insert_iterator(container);
}
```

istream_iterator

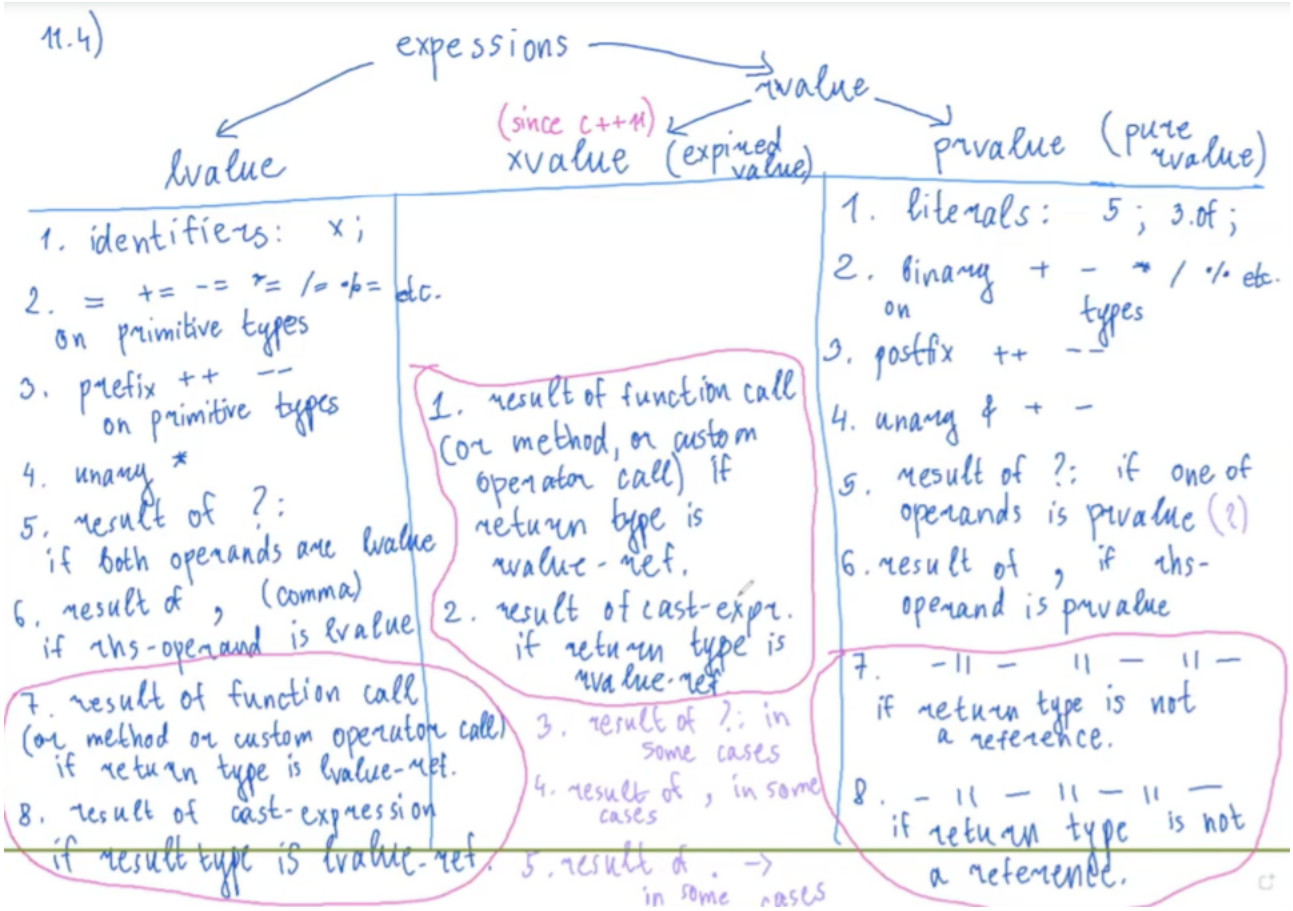
```
std::vector<int> v;

std::istream_iterator<int> it(std::cin);

for (int i = 0; i < 5; ++i, ++it) {
    v.push_back(*it);
}

for (int i = 0; i < 5; ++i) {
    std::cout << v[i] << ' ';
}
```

Зачем это надо? В стандартных алгоритмах, в которых требуются итераторы, не обязательно предоставлять итераторы на контейнеры, достаточно передать итератор на поток ввода

```
template <typename T>
class istream_iterator {
    std::istream& in;
    T value;
public:
    istream_iterator(std::istream& in): in(in) {
        in >> value;
    }

    istream_iterator<T>& operator++() {
        in >> value;
    }

    T& operator*() {
        return value;
    }
};

// std::ifstream in("input.txt");
// std::istringstream iss(s);
```

4.20 Формальное определение lvalue и rvalue. Объяснение идеи, стоящей за этим определением. Объяснение, как по выражению понять его вид value. Примеры, когда rvalue-выражение допускает присваивание и когда lvalue-выражение не допускает присваивания. Ссылочные квалификаторы (ref-qualifiers), синтаксис и пример использования.

Эти понятия применимы не к объектам и не к типам объектов, а к expressions.
Любой объект может быть как lvalue так и rvalue в зависимости от контекста, хотя опять-таки нельзя говорить про rvalue и lvalue относительно объектов. Интуитивно rvalue - выражение, которе представляет из себя создание нового объекта (то, что не может стоять справа от знака равенства). А lvalue - выражение, представляющее из себя обращение к уже существующему объекту (то, что может стоять слева от знака равенства). К данным определениям есть контр-примеры, это не всеобъемлющее определение.



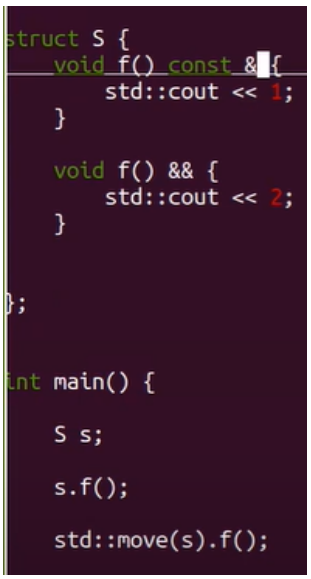
Примеры, когда rvalue-выражение допускает присваивание и когда lvalue-выражение не допускает присваивания.

```
int main() {
    int x = 0;
    int& rx = 2; // тут lvalue нельзя инициализировать rvalue
    int&& rrx = 1; // тут можно присвоить rvalue rvalue
}
```

Ссылочные квалификаторы (ref-qualifiers), синтаксис и пример использования

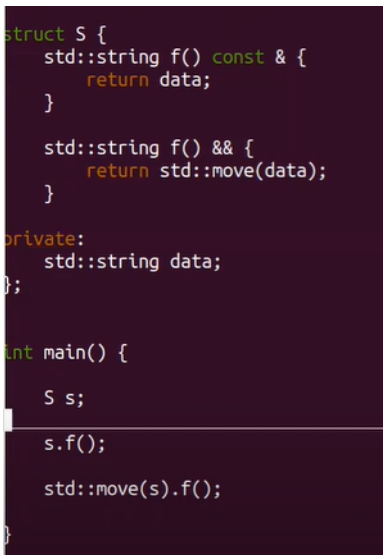
Аналогично константности, мы можем захотеть перегружать функции в зависимости от того, какое выражение является левым операндом функции - lvalue или rvalue. Если не ставим & то считается, что функция как для lvalue, так и для rvalue.

```
struct S{
    void f() &{...}
    void f() &&{...}
}
```



Сначала попадем в 1 версию, потом - во 2

Зачем эту нужно. Пусть у нас есть какой-нибудь метод, который возвращает какие-то данные. Если мы понимаем,что нам передали объект, у которого мы можем все забрать (наш объект временный и скоро будет уничтожен), тогда можем отправить его в соответствующую функцию



4.21 Rvalue-ссылки, их сходства и различия с обычными ссылками. Правила инициализации rvalue-ссылок. Примеры передачи параметров по rvalue-ссылке, перегрузка между обычными и rvalue-ссылками, логика выбора версии функции в случае такой перегрузки. Правила сворачивания ссылок (reference collapsing).

Неконстантная lvalue reference позволяет себя инициализировать только значениями lvalue, а **константные lvalue reference** значениями и lvalue и rvalue.

Rvalue reference позволяет себя инициализировать только rvalue-выражениями.

```
int x = 7;
int &lref = x; // инициализация ссылки l-value переменной x (значение l-value)
int &&rref = 7; // инициализация ссылки r-value литералом 7 (значение r-value)
```

Рассмотрим как работают rvalue references

```
int x = 0;
int& rx = x;
int& rx = 1; // WRONG! - надо присваивать lvalue
const int& crx = 1; // ок - это const ref и ей можно присваивать rvalue

int&& rrx = 1; // ок - присваиваем rvalue
int&& rrx = x; // WRONG! - надо присваивать rvalue

// Как создать rvalue из чего угодно? А вот так:
int&& ref1 = std::move(x); // This works fine

rrx = x; // ок - мы не инициализируем, а присваиваем значение из x
rrx = 5; // ок - аналогично

// при этом заметим:
x = 1; // rrx is still 5;
int& another_ref = rrx; // ок - rrx это идентификатор (т.е. lvalue)
int&& another_ref_2 = rrx; // WRONG! - потому что rrx не rvalue
```

В строчках 4 и 6 происходит продление жизни объекта, ссылка создается и будет жить на стеке пока не выйдет из области видимости. При этом переменная может сразу принять ислам, как если бы мы делали `String&& ref = String("aaaa")`. Строка ”aaaa” удаляется сразу после инициализации ссылки, а ref продолжает быть валидным и указывать на ”aaaa”.

Передача параметров по ссылкам

```
void fun(const int &lref) { // перегрузка функции для работы с l-values
  std::cout << "l-value reference to const\n";
}

void fun(int &&rref) { // перегрузка функции для работы с r-values
  std::cout << "r-value reference\n";
}

int main() {
  int x = 7;
  fun(x); // аргумент l-value вызывает функцию с ссылкой l-value
  fun(7); // аргумент r-value вызывает функцию с ссылкой r-value
}
```

Замечание: Если есть конструктор от const lvalue-ссылки и от rvalue-ссылки, компилятор отдаст rvalue объект во второй конструктор, так как **сработает перегрузка.** Хотя обе функции вроде бы подходят, второй случай считается perfect match

Правила сворачивания ссылок (reference collapsing):

- & + & = &
- & + && = &
- && + & = &
- && + && = &&

Свертывание ссылок происходит в контекстах, таких как

- инстанцирование шаблона
- определение auto переменных

Рассмотрим данный фрагмент кода:

```
template <typename T>
void func(T t) {
  T& k = t;
}

int main() {
    int i = 4;
    func<int&>(i);
}
```

При инстанцировании шаблона T установится равным int&. Какой же тип будет у переменной k внутри функции? Компилятор «увидит» int& & – а так как это запрещенная конструкция, компилятор просто преобразует это в обычную ссылку по правилу сворачивания ссылок.

4.22 Идея универсальных ссылок. Реализация функции std::move. Объяснение действия этой функции. Объяснение, почему принимаемый и возвращаемый типы именно такие.

Некоторые функции должны уметь принимать в качестве параметров как lvalue-reference, так и rvalue-reference. Здесь к нам на помощь приходят **universal references**. Универсальная ссылка обязана быть шаблонным параметром функции.

```
template <class T>
    void func(T&& t) {
}

func(4);              // 4 это rvalue: T становится int

double d = 3.14;
func(d);              // d это lvalue; T становится double&

float f() {...}
func(f());            // f() это rvalue; T становится float
```

Пояснение: Если была передана lvalue типа U, то T становится U& и `decltype(t) = int&` Если же U это rvalue, то T становится просто U и `decltype(t) = int&&`

Реализация std::move

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

Функция принимает универсальную ссылку так как задача move: превратить все (а мы не знаем что именно нам передали) в rvalue. Для возвращаемого типа используем type_traits, и поэтому делаем каст нашего типа к rvalue.

Замечание: `std::move` надо писать когда хотим из lvalue сделать rvalue, для rvalue объектов все и так будет работать правильно

4.23 Проблема прямой(идеальной) передачи. Предназначение функции emplace_back, ее преимущество перед push_back. Правильное использование функции std::forward (без реализации) для реализации механизма perfect forwarding.

Проблема: Когда у нас в шаблонную функцию передается пременное число аргументов, и мы не знаем какие из них rvalue, а какие lvalue, встает вопрос как передать дальше как rvalue те и только те, что изначально были rvalue?

Решение: Можем воспользоваться функцией std::forward

Пример **(механизм perfect forwarding)**:

```
template <typename ...Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Теперь все типы, кототые были переданы в f как lvalue будут иметь тип `type&`, а которые были переданы как rvalue - `type&&`. Пусть внутри функции f вызывается функция g, принимающая пакет аргументов. Хотелось бы применить std::move к тем, которые являются rvalue reference и скорпировать остальные (к ним нельзя применять std::move, так как их передали в f не как rvalue reference, следовательно не ожидают удаления всей инфорации). Поэтому применяем std::forward.

Реализация std::forward

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

Такая конструкция породит rvalue для объектов которые были изначальное отданы как rvalue и lvalue для всех остальных, это значит все аргументы проходят в функцию g с такими же видами value с какими нам их дали, как следствие сможем копировать только те объекты, которые нам изначально пришлось бы копировать.

push_back и emplace_back

Note: emplace_back есть во всех контейнерах, в котрорых есть push_back. Для контейнеров, в которых есть только insert, есть аналогичная функция emplace.

Решение проблемы с push_back в векторе (см билет 3.30) воплощено в функции emplace_back, которая семантически аналогична push_back. Однако в emplace_back не создается промежуточного временного объекта, так как в construct передаем сразу (args...), где args... - аргументы добавляемого объекта, которые **пробросятся сразу в конструктор**, и таким образом временная строка (которая потом бы скопировалась) не будет создана. То есть строка создается единственный раз и сразу на нужном месте.

```
template<typename... Args>
void emplace_back(const Args&... args){
    if(sz == cp) reserve(2 * cp);
    AllocTraits::construct(alloc, arr + sz, args...);
}
```

Преимущество emplace_back над push_back

```
std::vector<std::string> v;

v.push_back(std::string("abc")); // дважды создается строка
// первый - для передачи в функцию, второй - создается тот,
// который будет лежать в векторе (копируется в constuct)

v.emplace_back("abc"); // создается единожды
// сразу на нужном месте с нужным значением
```

На самом деле, emplace_back не решает проблему с излишним копированием, а только переносит ее на другой уровень. Ведь если среди аргументов, которые мы передаем в конструктор так же будут нетривиальные для копирования объекты, то в результате того, что мы передаем эти аргументы по констатнтой ссылке, будут вызваны лишние копирования.

4.24 Контейнер vector: внутреннее устройство с алгоритмической точки зрения, методы reserve, capacity, shrink_to fit, их действие. Проблема реализации выделения памяти в методе reserve. Реализация методов reserve, resize и push_back с использованием аллокатора (здесь можно без поддержки exception safety). Правила инвалидации итераторов в vector.

Устройство вектора:

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
private:
    T* arr;
    size_t sz;
    size_t capacity;
    Alloc alloc;

    using Alloctraits = std::alloctator_traits<Alloc>;

public:
    Vector(size_t n, const T& value = T(), const Alloc& alloc = Alloc());

    T& operator[](size_t i) {
        return arr[i];
    } //also this method but for const Vector

    T& at(size_t i) {
        if (i >= sz) throw std::out_of_range("...");
        return arr[i];
    } //also this method but for const Vector

    size_t size() const {
        return sz;
    }

    size_t capacity() const {
        return cap;
    }

    void resize(size_t n, const T& value = T());
    void reserve(size_t n);
};
```

Замечание: При использовании в векторе типа без конструктора по умолчанию, мы обязаны при инициализации указывать тогда каким значением проинициализировать ячейки

В чем отличие resize от reserve?

- Resize - выделяет памяти столько, чтобы ее хватило на n элементов, т.е меняет размер.
- Reserve - меняет capacity. Обычно вектор не уменьшает capacity, чтобы потом не перевыделять память снова. Но если хотим уменьшить capacity до текущего размера, то можно вернуть системе с помощью вызова `shrink_to_fit()`

Рассмотрим реализацию метода reserve() поэтапно:

Этап 1: В коде ниже представлена плохая реализация reserve(). Почему она плохая? У нас фактически reserve() работает как resize(), что плохо, а мы просто должны выделять память на n объектов, а не заполнять их значениями по умолчанию (просто потому, что конструктора по умолчанию типа T может просто не быть).

```
void reserve(size_t n) {
    if (n <= cap)
        return;

    T* newarr = new T[n];
    for (size_t i = 0; i < sz; ++i) {
        newarr[i] = arr[i];
    }

    delete[] arr;
    arr = newarr;
}
```

Этап 2: Более приемлемая реализация уже выделяет необходимое количество байт для хранения. Но мы не можем делать присваивание к newarr[i], так как в реальности под newarr[i] лежит сырая память ⇒ будет SegFault. Таким образом, нам нужно вызвать конструктор T по данному адресу от данного объекта.
Для этого существует специальный синтаксис: **placement-new**. (смотри строку 7)
Но проблема не устранена, так как delete[] тоже будет SegFault, так как в arr в реальности лежит sz объектов, а не cap, то есть часть объектов – это сырая память. (Решим эту проблему на следующем этапе с помощью вызова деструктора вручную для каждого объекта и потом возврата системе сырой памяти)

```
void reserve(size_t n) {
    if (n <= cap)
        return;

    T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
    for (size_t i = 0; i < sz; ++i) {
        new(newarr + i) T(arr[i]);
    }

    delete[] arr;
    arr = newarr;
}
```

Этап 3: Корректная реализация с помощью *uninitialized_copy*, которая безопасна относительно исключений.

```
void reserve(size_t n) {
    if (n <= cap) return;

    T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
    std::uninitialized_copy(arr, arr + sz, newarr);

    for (size_t i = 0; i < sz; ++i) {
        (arr + i)->~T();
    }
    delete[] reinterpret_cast<int8_t*>(arr);
    arr = newarr;
}
```

При этом сам *uninitialized_copy* реализован так:

```
////// uninitialized_copy realization
size_t i = 0;
try {
    for (; i < sz; ++i) {
        new(newarr + i) T(arr[i]);
    }
} catch(...) {
    for (size_t j = 0; j < i; ++j) {
        (newarr + j)->~T();
    }
    delete[] reinterpret_cast<int8_t*>(newarr);
} //////
```

Этап 4: Копирование – это плохо и неэффективно. Можно сделать умнее! Самая хорошая реализация с помощью Allocator & std::move

```
void reserve(size_t n) {
    if (n <= cap) return;

    T* newarr = AllocTraits::allocate(Alloc, n);

    size_t i = 0;
    try {
        for (; i < sz; ++i) {
            AllocTraits::construct(alloc, newarr + i, std::move(arr[i]));
        }
    } catch(...) {
        for (size_t j = 0; j < i; ++j){
            AllocTraits::destroy(alloc, newarr + j);
        }
        AllocTraits::deallocate(newarr, n);
        throw;
    }

    for (size_t i = 0; i < sz; ++i) {
        AllocTraits::destroy(alloc, arr + i);
    }
    AllocTraits::deallocate(arr, n);
    arr = newarr;
}
```

Важно понимать, что **memcpy использовать нельзя**, так как у нас производный тип, а копирование может быть не тривиальным, например, если объект хранит ссылки, то при копировании ссылки могут начать указывать не туда.

Реализация остальных методов vector'a:

```
void push_back(const T& value) {
    if (sz == cap)
        reserve(2 * cap);

    //new(arr + sz) T(value);
    AllocTraits::construct(alloc, arr + sz, value);
    ++sz;
}

void pop_back(const T& value) {
    //(arr + sz - 1)-> T();
    AllocTraits::destroy(alloc, arr + sz - 1);
    --sz;
}

void resize(size_t n, const T& value = T()) {
    if (n < cap) reserve(cap);
    /*...*/
}
```

Инвалидация итераторов

Допустим у нас есть вектор и свободное пространство рядом с ним. Если cap закончилось, то вектор реаллоцирует свой storage: перекладываем все элементы и уничтожаем старый. Что если у нас был итератор на старый вектор??? После перекладывания наш итератор инвалидировался. Если теперь обратиться по итератору, то это UB. Так же push_back инвалириует обычные указатели и ссылки на элементы вектора.

```
vector<int> v;
v.push_back(1); // sz=1, cap=1
vector<int>::iterator it = v.begin();
int* p = &v.front();
int& r = v.front();
v.push_back(2); // sz=2, cap=2
//storage reallocated - so we face invalidation
```

4.25 Контейнер vector<bool>. Отличие от обычного vector. Класс BoolReference, его реализация. Реализация метода [] в vector<bool>.

Он отличается от обычного вектора тем, что хранит не просто массив буллей, а пакует его в **пачки по 8 логических значений и представляет их как один байт.** (То есть на одно значние приходится 1 бит)

В vector<bool> интересно работает присваивание.

```
templtate <typename U>
void f(const U&) = delete;

int main() {
    vector<bool> vb(10, false);
    vb[5] = true;
    f(vb[5]);
}
```

В данном случае компилятор начнет ныть, что нельзя вызывать f от типа, который удален. Но так мы заставим компилятор спалить какой у него тип для vb[5].
Мы увидим, что **U = std::_Bit_reference**. Как же это работает?

```
templtate <>
class Vector<bool> {
    int8_t* arr;
    size_t sz;
    size_t cap;

    struct BitReference {
        int8_t* cell;
        uint8_t num; // pos in this cell

        BitReference& operator=(bool b) {
            if (b) {
                *cell |= (1u << num);
            } else {
                *cell &= ~(1u << num);
            }
            return *this;
        }

        operator bool() const {
            return *cell & (1u << num);
        }
    }

public:
    BitReference operator[](size_t i) {
        return BitReference{arr + i / 8, i % 8};
    }
}
```

Структура BitReference такаия хитрая, что она позволяет, присваиваяя экземпляру себя, менять исходный вектор.

4.26 Контейнеры list и forward_list, их внутреннее устройство, примерная реализация основных методов (конструкторы, деструкторы, операторы присваивания, insert, erase, push/pop_back, push/pop_front). Правила инвалидации итераторов в list и forward_list.

std::list представляет собой контейнер, который поддерживает быструю вставку и удаление элементов из любой позиции в контейнере. Быстрый произвольный доступ не поддерживается. Он реализован в виде двусвязного списка. В отличие от **std::forward_list** этот контейнер обеспечивает возможность двунаправленного итерирования, являясь при этом менее эффективным в отношении используемой памяти.

forward_list - односвязный список,в котором отсутствует методы *back*(), *pop_back*()

Об устройстве:

Связный список aka **list**, хранит внутренний тип *Node* для хранения "вершинок" списка. Ноды хранят элемент листа и указатели на предыдущую и следующую вершинку (логично, что в **std::forward_list** хранится указатель только на следующую вершинку).

В полях листа хранится указатель на начало списка, размер листа и аллокатор.

Хранить список можно так: Делаем фейковую вершинку – **head**. Ссылкой next она указывает на первую ноду list'а, а ссылкой prev – на последнюю вершинку. Итератор указывает на *Node*.

```
template<typename T, typename Alloc = std::allocator<T>>
class List {
private:
    struct Node {
        T val;
        Node* next = nullptr;
        Node* prev = nullptr;
        explicit Node(const T& val) : val(val) {}
    };

    size_t sz = 0;
    Alloc t_allocator;
    typename Alloc::template rebind<Node>::other allocator;
    Node* head;

public:

    ////////////// ITERATORS ////////////
    template <bool IsConst>
    class common_iterator {
    public:
        std::conditional_t<IsConst, const Node*, Node*> ptr_node;
        /*.....*/
    };

    using t_node_alloc = std::allocator_traits<Alloc>;
    using node_alloc = std::allocator_traits
                            <typename Alloc::template rebind<Node>::other>;
    using iterator = common_iterator<false>;
};
```

Реализация основных методов

```
////////////// LIST METHODS ////////////
explicit List(const Alloc& alloc = Alloc()) : sz(0), t_allocator(alloc) {
    head = node_alloc::allocate(allocator, 1);
    head->next = head;
    head->prev = head;
}
explicit List(size_t n, /*const T& val,*/ const Alloc& alloc = Alloc()) : List(
    alloc) {
    if (n < 0) throw std::bad_alloc();
    size_t i = 0;
    try {
        for (; i < n; ++i) {
            Node *new_node = node_alloc::allocate(allocator, 1);
            /*node_alloc::construct(allocator, new_node, val);*/
            node_alloc::construct(allocator, new_node);
            link_nodes(head->next, new_node);
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            Node* copy_node = head->prev;
            del_node(head->prev);
            node_alloc::destroy(allocator, copy_node);
            node_alloc::deallocate(allocator, copy_node, 1);
        }
        node_alloc::deallocate(allocator, head, 1);
        throw;
    }
    sz = n;
}

List(const List&& l) : sz(l.sz),
t_allocator(t_node_alloc::select_on_container_copy_construction(l.t_allocator)),
allocator(node_alloc::select_on_container_copy_construction(l.allocator))
{
    Node *new_head = node_alloc::allocate(allocator, 1);
    new_head->next = new_head;
    new_head->prev = new_head;

    Node* l_begin = l.head->next;
    size_t i = 0;
    try {
        for (; i < l.sz; ++i) {
            Node* new_node = node_alloc::allocate(allocator, 1);
            node_alloc::construct(allocator, new_node, l_begin->val);
            link_nodes(new_head->prev, new_node);
            l_begin = l_begin->next;
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            Node* copy_node = new_head->prev;
            del_node(new_head->prev);
            node_alloc::destroy(allocator, copy_node);
            node_alloc::deallocate(allocator, copy_node, 1);
        }
        node_alloc::deallocate(allocator, new_head, 1);
        throw;
    }
    sz = l.sz;
```

```
    head = node_alloc::allocate(allocator, 1);
    head = new_head;
}

~List() {
    while (sz != 0)
        pop_back();
    node_alloc::deallocate(allocator, head, 1);
}

List& operator=(const List& l) {
    while (sz != 0)
        pop_back();
    if (node_alloc::propagate_on_container_copy_assignment::value) {
        node_alloc::deallocate(allocator, head, 1);
        allocator = l.allocator;
        t_allocator = l.t_allocator;
        head = node_alloc::allocate(allocator, 1);
        head->next = head;
        head->prev = head;
    }
    Node* l_begin = l.head->next;
    for (size_t i = 0; i < l.sz; ++i) {
        push_back(l_begin->val);
        l_begin = l_begin->next;
    }
    return *this;
}
```

Реализация вставки и удаления

```
void push_back(const T &val) { insert(end(), val); }
void push_front(const T &val) { nsert(begin(), val); }
void pop_back() { erase(--end()); }
void pop_front() { erase(begin()); }

template<bool IsConst>
iterator insert(const common_iterator<IsConst> &it, const T &val) {
    Node* new_node = node_alloc::allocate(allocator, 1);
    try {
        node_alloc::construct(allocator, new_node, val);
        link_nodes((it.ptr_node)->prev, new_node);
    } catch (...) {
        Node* copy_node = new_node;
        del_node(new_node);
        node_alloc::destroy(allocator, copy_node);
        node_alloc::deallocate(allocator, copy_node, 1);
        throw;
    }
    ++sz;
    return List::iterator(new_node);
}
template<bool IsConst>
iterator erase(const common_iterator<IsConst> &it) {
    Node* copy_node = const_cast<Node*>(it.ptr_node);
    iterator copy_it(copy_node);
    del_node(it.ptr_node);
    node_alloc::destroy(allocator, it.ptr_node);
    node_alloc::deallocate(allocator, copy_node, 1);
    --sz;
    return ++copy_it;
}
```

Среди методов list - *sort* (так как stl-евская std::sort работает на RA-итераторах, в листе реализована сортировка слиянием), *reverse*, *merge*, *splice* (двух списком целиком или часть одного списка вклеить в другой список)

Инвалидация итераторов

list: Никакой из методов `insert, emplace_front, emplace_back, emplace, push_front, push_back` **не инвалидирует** итераторы и ссылки.

forward_list: Ни одна из перегрузок метода `insert_after` **не инвалидирует** итераторы и ссылки.

4.27. Контейнеры map и set, их внутреннее устройство с алгоритмической точки зрения: что хранится внутри, какие алгоритмы и структуры данных используются для реализации методов. Понятие компараторов, пример использования нестандартных компараторов. Асимптотика обхода map итератором с объяснением, как он (обход) работает. Правила инвалидации итераторов в map.

Map - упорядоченный ассоциативный массив, который хранит пару: ключ-значение; в c++ это красно-черное дерево (сбалансированное двоичное дерево поиска). В нем тоже есть структура **Node**, которая хранит ключ-значение как пару, указатель на родителя, указатели на двух детей, bool red. Итератор в map - указатель на **Node**. Под итератором лежит пара ключ-значение.

Еще есть **компаратор** (для ключей) - функция, которой на вход можно передать два элемента для сравнения, а на выходе получить true или false в зависимости от результатов сравнения; по стандарту предполагается, что компаратор задаёт weak ordering на множестве (транзитивное бинарное отношение, которое для любых двух элементов либо xRy, либо yRx). Пример нестандартного компаратора - сравнение пар элементов.

Ключ обязан иметь **константный тип** (const key), так как от ключа зависит положение в дереве.

Если в insert положить пару с неконстантным key, то произойдёт const_cast; если положить такой элемент, что key уже существует, то bool в паре будет false. Если по [key] не существует такого элемента, то он создастся по умолчанию, а если через at - то бросает исключение.

От map нельзя вызывать std::sort, next_permutation - CE.

Структура	Параметры	Описание
map	<Key, Value, Comp = std::less<Key> >	Красно-чёрное дерево
unordered_mop	<Key, Value, Hash = std::hash<Key>, EqualTo = std::equal_to <Key> >	Хеш-таблица, массив связных списков
set	<Key, Compare = std::less<Key> >	Красно-чёрное дерево

Name	iterator find	pair<iterator, bool> insert
map	(const Key&), O(log N)	(const pair<const Key, Value>&), O(log N)
set	O(log N)	O(log N)

Name	bool erase	Value& operator[]	Value& at
map	(const Key&), O(log N)	(const Key&), O(log N)	(const Key&), O(log N)
set	O(log N)	-	-

Обход map итератором:

Строго говоря, инкремент в итераторе map работает за логарифм (как и find по ключу), но в итоге обход мап итератором (см. код) линеен:

```
for (auto it = m.begin(); it != m.end(); ++it)
  cout << it->first << it->second;
```

Он линеен, так как в итоге каждую вершину дерева мы посещаем не более шести раз: в неё входит одно ребро (от родителя), выходят два ребра (к детям), значит, по ней при обходе можно будет пройтись не более шести раз по этим рёбрам.

Пример оптимального использования итераторов в map:

```
// Неоптимально:

if (m.count(5))
  m[5] = 3;

// Неоптимально, так как мы два раза спустились по дереву
// Оптимально:

auto x = m.find(5);
if (x != m.end())
  x->second = 3;
```

Инвалидация итераторов в map:

Метод	Инвалидация
All read only operations, swap, std::swap	Никогда
clear, rehash, reserve, operator=	Всегда
insert, emplace, emplace_hint, operator[]	Только если был rehash
erase	Только относительно удалённого элемента

4.28. Мотивировка умных указателей. Класс unique_ptr, его идея и реализация основных методов. Особенности поведения unique_ptr при попытке его скопировать. Функция make_unique, её реализация, пример использования, преимущество перед обычным конструктором unique_ptr

Умный указатель - инструмент, который позволяет автоматически освобождать динмаически выделенные ресурсы. std::unique_ptr и std::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, давайте вспомним про RAII и исключения в конструкторах, функциях).

unique_ptr должен использоваться, когда ресурс памяти не должен быть разделяемым (у этого указателя нет к-ра копирования), но он может быть передан другому unique_ptr.

unique_ptr **нельзя копировать**, соответствующий конструктор и оператор у них удалены, но зато можно мувать (поэтому вектор из UP корректен - для этого же нужны noexcept'ы (они не обязательны, но нужны, чтобы вектор гарантировал безопасность исключений)). Присвоить значение этому указателю можно только в момент объявления (т.е. инициализация умного указателя должна быть в момент объявления).

```
template<typename T, typename Deleter = std::default_delete<T>>
class unique_ptr {
 private:
  T* pointer;
 public:
  unique_ptr() {
    pointer = nullptr;
  }

  explicit unique_ptr(T* ptr): pointer(ptr) {}

  unique_ptr(const unique_ptr<T>& other) = delete;

  unique_ptr<T>& operator=(const unique_ptr<T>& other) = delete;

  unique_ptr(unique_ptr<T>&& other)  noexcept {
    pointer = std::move(other.pointer);
    other.pointer = nullptr;
  }

  unique_ptr& operator=(unique_ptr<T>&& other)  noexcept {
    delete pointer;
    pointer = std::move(other.pointer);
    other.pointer = nullptr;
  }

  ~unique_ptr() {
    delete pointer;
  }

  T& operator*() const {
    return *pointer;
  }

```

```
};
```

На самом деле, unique_ptr принимает два шаблонных параметра: второй—это typename Deleter, у которого определен оператор () и он вызывается (как функция) в деструкторе (дефолтный Deleter вызывает delete). Unique_ptr - легковесный, быстрый; первый пример класса, который мувать можно, но копировать нельзя.

make_unique

Какие у нас есть проблемы с unique_ptr?

1. Мы не можем полностью избавиться от использования new (см. сравнение).
2. Нужно явно перечислять аргументы типа шаблона.
3. Exception safety: см. код.

```
// Сравнение make_unique и unique_ptr
std::make_unique<int>(1);
std::unique_ptr<int>(new int(1));

// Применение make_unique

class Fraction
{
 private:
  int m_numerator = 0;
  int m_denominator = 1;

 public:
  Fraction(int numerator = 0, int denominator = 1) :
           m_numerator(numerator), m_denominator(denominator)
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
  }
};

// exception safety:
void function(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B())) {
    ... }

void function(std::make_unique<A>(), std::make_unique<B>()) { ... }

// если B() создаёт исключение, а A - нет, то стандарт c++ не требует, чтобы
// первый объект был уничтожен; во втором же случае у нас временные объекты
// и стандарт c++ обязывает их уничтожить

int main()
{
  // Создаем объект с динамически выделенным Fraction с numerator = 7 и denominator = 9
  std::unique_ptr<Fraction> f1 = std::make_unique<Fraction>(7, 9);
  std::cout << *f1 << '\n'; // вывод: 7/9

  // Создаем объект с динамически выделенным массивом Fraction длиной 5.
  // Используем автоматическое определение типа данных с помощью ключевого слова auto
  auto f2 = std::make_unique<Fraction[]>(5);
  std::cout << f2[0] << '\n'; // вывод: 0/1

  return 0;
}
```

4.29. Идея реализации класса shared_ptr (без поддержки weak_ptr, без поддержки нестандартных аллокаторов и deleter'ов): конструкторы, деструктор, операторы присваивания. Функция make_shared, ее реализация, пример использования, преимущества перед обычным конструктором shared_ptr.

Если нам нужно иметь несколько указателей на один и тот же объект, то для этого воспользуемся **std::shared_ptr**. Внутри shared_ptr есть счетчик, который показывается, сколько копий у этого указателя существуют и указывают на то же что и он (решает проблему многократного удаления по одному и тому же указателю).

В этом случае оба умных указателя в равной мере управляют обычным указателем. Освобождение памяти произойдет в момент, когда последний shared_ptr, обладающий общим ресурсом, покинет область видимости. Оператор и конструктор копирования разрешены. shared_ptr предоставляет больше возможностей, но увеличивается и расход памяти, и время доступа.

```
template<typename T>
class shared_ptr {
 private:
  // на самом деле, не очень эффективно хранить ptr и counter как 2 отдельных указателя
  // отсюда ускорение при использовании make_shared, так как
  // при его использовании эти два указателя будут лежать рядом
  T* ptr = nullptr;
  size_t* counter = nullptr;
 public:
  // shared_ptr(T* ptr): ptr(ptr)
  // shared_ptr(shared_ptr other): ptr(other.ptr), count(++other.count)
  // реализация сверху НЕ ОК, т.к. никто не гарантировал, что shared ptr'ов 2

  // static count тоже не работает: он получится один на ВЕСЬ класс
  // вне зависимости от T

  // правильное решение: иметь указатель на счётчик
  shared_ptr() {}

  shared_ptr(T* ptr): ptr(ptr), counter(new size_t(1)) {}

  shared_ptr(const shared_ptr& other): ptr(other.ptr), counter(other.counter
   ) {
    ++*counter;
  }

  shared_ptr(shared_ptr&& other): ptr(other.ptr), counter(other.counter) {
    other.ptr = nullptr;
    other.counter = nullptr;
  }

  // + операторы присваивания, но с ними понятно

  // деструктор
  ~shared_ptr() {
    if (*counter > 1) {
      --*counter;
      return;
```

```
    }
    delete ptr;
    delete counter;
  }

  // разыменование
  T& operator*() const { return *ptr; }
  T& operator->() const { return ptr; }

  size_t use_count() const { return *count; }

};
```

Аргументация для **make_shared** совпадает с аргументацией для 4.28 (не использовать больше никогда new/delete, например), но теперь здесь добавляется скорость работы программы (см. комментарий в коде). Пишем make_shared!

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


template <typename T, typename ...Args>
shared_ptr<T> make_shared(Args&& ...args) {
  auto p = new ControlBlock<T>(1, std::forward<Args>(args)...);
  return p;
}

int main(){
  auto p = std::make_unique<int>(5); //copy elision called, no copy-ctor here
```

Пример использования:

```
// Пример использования
class Item
{
 public:
  Item() { std::cout << "Item acquired\n"; }
  ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
  // Выделяем Item и передаем его в std::shared_ptr
  auto ptr1 = std::make_shared<Item>();
  {
    auto ptr2 = ptr1; // создаем ptr2 из ptr1, используя семантику копирования

    std::cout << "Killing one shared pointer\n";
  } // ptr2 выходит из области видимости здесь, но ничего больше не происходит

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 выходит из области видимости здесь, и выделенный Item также уничтожается здесь
```

4.30 Проблемы, приводящие к идее автоматического вывода типов (auto) при объявлениях. Примеры, когда auto необходимо и когда неуместно. Ключевое слово decltype. Особенности поведения auto и decltype от ссылочных типов. Особенности auto в возвращаемом типе функции. Особенности поведения decltype от lvalue, xvalue и prvalue. Конструкция decltype(auto) и пример, когда она нужна.

Мотивировка auto

Прописывать тип целиком неудобно, код становится нечитаемым.

Решение этой проблемы - ключевое слово auto, которое сообщает компилятору, что тип переменной должен быть установлен исходя из типа инциализируемого значения.

Оно работает почти так же как вывод шаблонного типа, компилятор смотрит на тип выражения и подставляет нужный на этапе компиляции. auto работает по тем же правилам, то есть если исходный тип должен быть ссылкой, то и auto надо писать с ссылкой, аналогично с константностью. auto следует понимать аналогично T - это универсальная ссылка.

Примеры, когда auto уместно:

```
int main() {
    unordered_mao<string, int, hash<string>, equal_to<string>, FastAllocator
   <pair<string, int>> m;
    auto mm = m;
    return 0;
}
```

Забыв const, которое закомичено, мы получим копирование на каждом шаге, auto позволяет этого избежать:

```
#include <iostream>

int main() {
    for(const pair</*const*/string, int>& item: mm) {}
    for(auto it = begin(); it != end(); ++it) {}
    for(const auto& item: mm) {}
    //или
    return 0;
}
```

В данном случае это проблема, потому что map по реализации хранит const ключ.

Примеры, когда auto неуместно:

Когда вы пишете auto, вы предполагаете, что следующие строчки эквивалентны:

```
std::vector<bool> v(5, true);
bool bit = v[2]; // явно достаем значение из вектора в bool
auto bit = v[2]; // UB - получим тип std::vector<bool>::bit_reference
```

Но это не так. В коде, использующем auto, тип bit больше не является bool. Хотя концептуально vector<bool> хранит значения bool, operator [] у std::vector<bool> не возвращает ссылку на элемент контейнера (то, что std::vector::operator[] возвращает для всех типов за исключением bool). Вместо этого возвращается объект типа std::vector<bool>::reference (класса, вложенного в std::vector<bool>).

Ключевое слово decltype

В компайл-тайм возвращает тип выражения. Хорош тем, что не отбрасывает &&, &, *, const/volatile(ключевое слово, которое говорит компилятору не оптимизировать), как это делается при принятии аргументов в функцию. С помощью decltype можно отличить ссылку на объект от исходного объекта. На decltype тоже можно навесить модификаторы типа, при навешивании ссылок на decltype, в которм уже есть ссылки произойдет сворачивание ссылок.

Выражение внутри decltype никогда не выполняются, только оценивается их тип, так как это все происходит на этапе компиляции.

```
int x = 7;
decltype(x++) u = x;
```

Если внутри decltype выражение типа prvalue типа T, то тип declype(expression) - T
Если внутри decltype выражение типа xvalue типа T, то тип declype(expression) - T&&
Если внутри decltype выражение типа lvalue типа T, то тип declype(expression) - T&

Type deduction for return type

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

```
auto f(int& x){
    if (x > 5) return x;
    else return 0.0;
} // CE, inconsisent deduction
```

Другой пример: пусть функция возвращает функцию. Функция $f(x)$ может возвращать как по ссылке так и по значению, и мы не знаем этого заранее. Написать auto(f(x)) нельзя, так как на данном этапе x еще не определен. Решение проблемы: trailing return type:

```
auto g(int& x) -> decltype(f(x)){
    return f(x);
}
```

С C++14 возможен вот такой синакисис, который говорит компилятору: выведи тип самостоятельно, но не по правилам auto, а по правилам decltype:

```
template <typename Container>
decltype(auto) g(const Container& cont, size_t index)){
    std::cout << "...";
    return cont[index];
}
```

4.31. Лямбда-функции и лямбда-выражения, их синтаксис. Пример использования в стандартных алгоритмах, пример использования в качестве компаратора для map. Списки захвата в лямбда-выражениях, их синтаксис, пример использования. Синтаксис захвата по ссылке и по значению.

С C++11 появились лямбда-функции. Они помогают описывать функции прямо внутри выражения, которое их вызывает.

До:

```
struct MyCompare {
    bool operator()(int x, int y) const {
        return std::abs(x - 5) < std::abs(y - 5);
    }
};

std::vector<int> v{1, 5, 4, 7};
std::sort(v.begin(), v.end(), MyCompare());
```

Основное неудобство - приходится объявлять функцию/компаратор снаружи области видимости - нарушение инкапсуляции.

После, синтаксис:

```
std::sort(v.begin(), v.end(),
    [](int x, int y) {
        return std::abs(x - 5) < std::abs(y - 5);
    });
```

Объявление [] - **closure expression**, дальше параметры, дальше тело функции. Можно объект проинициализировать лямбда-функцией, тип объекта - ожидаемо auto.

Можно возвращать функции из других функций (пишем лямбда функцию после return), в примере ниже написан компаратор, который можно передавать в качестве параметра сортировки)

```
    auto getCompare() {
        return [](int x, int y) {
            return std::abs(x - 5) < std::abs(y - 5);
        }
    }

    [](int x) {
        std::cout << x << "\n"; //declaration, rvalue
    };

    [](int x) {
        std::cout << x << "\n"; //called
    }(5);
```

Тип возвращаемого значения определяется по правилам вывода типов, он установлен по умолчанию. Если так получилось, что компилятор сам не справляется (например две ветки условий, и в каждой возвращаемое значение разниое) или мы хотим кастомный type_deduction, то можно в явном виде прописать, какого типа вывод мы ожидаем

```
    [](int x) -> bool {
        std::cout << x << "\n"; //declaration, rvalue
    };
```

Capture lists

```
int a = 1;

[](int x) {
   std::cout << x + a << "\n";
}(5);
```

Будет CE потому что a не захвачен внутрь лямбда-функции, локальные объекты не попадают в область видимости, в отличие от глобальных или конкретных namespace.

Надо было написать так:

```
int a = 1;

[a](int x) {
   std::cout << x + a << "\n";
}(5);
```

Но так нельзя будет менять переменную a - ее тип const int. Чтобы можно было менять надо написать так: `[a](int x) mutable {...};`

```
[a](int x) mutable {
   std::cout << x + a << "\n";
   ++a;
}(5);
```

Однако здесь тип a - int& !

Лямбда-функция для map:

```
auto f = [](int x, int y){...};
std::map<int, int, decltype(f)> m;
```

4.32 Идиома SFINAE. Простейший пример выбора из двух функций с использованием SFINAE (когда более подходящую функцию пришлось отбросить). Структура enable_if, ее реализация и пример правильного использования.

Идиома SFINAE.

SFINAE - правило вывода шаблонных версий функций.
Substituon Failure Is Not An Error - неудачная шаблонная подстановка не является ошибкой компиляции.

В данном примере мы не смогли подставить T = int, так как у данного типа нет метода size(). Вместо CE компилятор отбрасывает верхнюю версию и идёт искать дальше подходящую, это и есть SFINAE:

```
template <typename T>
auto f(const T&) -> decltype(T().size()) {
    std::cout << 1;
}

void f(...) {
    std::cout << 2;
}

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

Работает для сигнатуры, не работает для тела функции.

Структура enable_if

Структура, которая позволяет выбирать перегрузки функций в зависимости от compile-time проверяемых условий

Реализация enable_if

```
template <bool B, typename T = void>
    struct enable_if{};

template <typename T>
struct enable_if<true, T>{
    using type = T;
};

template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

Использование enable_if

```
template<typename  T, typename = std::enable_if_t<std::is_class_v<T>>>
void g(const T&) {
        std::cout << 1;
}

template<typename T, typename = std::enable_if_t<!std::is_class_v<
```

```
    std::remove_reference_t<T>>>>
    void g(T&&) {
            std::cout << 2;
    }
```

4.33 Константные выражения, ключевое слово constexpr. Отличие constexpr от const. Контексты, в которых требуются константные выражения. constexpr-функции и ограничения на их содержимое. Особенности throw в constexpr-функциях, особенности создания объектов в constexpr-функциях. Ключевое слово static_assert и его применение. Пример ситуации, когда static_assert неприменим.

Некоторые контексты требуют, чтобы переменная была известна на этапе компиляции.

Отличие constexpr от const

Если переменная является константной, это ещё не значит, что она compile-time вычисляемая:

```
int x;
cin >> x;
const int y = x;
```

Контексты, в которых требуются константные выражения

Constexrt - тип выражения, который является compile-time вычислимым. Такие выражения можно делать шаблонными параметрами:

```
constexpr int y = x;
std::array<int, y> arr;
```

Второй контекст, в котором это может потребоваться - инициализация статической константы: static const int y = constexpr-type expr.

Constexpr-функции и ограничения на их содержимое

Рассмотрим пример, который не скомпилируется:

```
/*constexpr*/ int factorial(int n) {
    if (n==0) return 1;
    return n * factorial(n-1);
}

int main() {
    constexpr int y = factorial(5);
    return 0;
}
```

Вызов функции должен быть константным выражением. Чтобы сделать его таковым, раскомментируем constexpr.

Constexpr-функции - такие, которые могут быть исполнены в compile-time.

Ограничения на содержимое:

1. Не виртуальна (до 20 стандарта)
2. Возвращаемый тип - литеральный
3. Каждый параметр - литеральный тип

4. Для конструктора (после 20 стандарта): класс не должен иметь виртуальный базовый класс

5. Тело функции не должно быть function-try-block

Литерал — это элемент программы, который непосредственно представляет значение. Целочисленные литералы начинаются с цифры и не имеют дробных частей или экспонент. Литералы с плавающей запятой задают значения, которые должны иметь дробную часть. Эти значения содержат десятичные разделители (.) и могут содержать экспоненты и далее, например, логические литералы true & false.

Особенности throw в constexpr функциях, особенности создания объектов в constexpr функциях

В constexpr функциях нельзя бросать исключения.

В общем случае, нельзя создавать объекты на этапе компиляции, но если конструктор помечен constexpr, то можно.

Если функция помечена constexpr это означает, что она **может** быть вычислена на этапе компиляции, но если мы вызываем её в неподходящем для этого контексте, она будет вызвана уже в runtime.

Ключевое слово static_assert и его применение

Проверяет программное утверждение во время компиляции. Если указанное константное выражение имеет значение false , компилятор отображает указанное сообщение, если оно предоставлено, и компиляция завершается ошибкой; в противном случае объявление не оказывает никакого влияния.

```
static_assert( constant-expression, string-literal );
static_assert( constant-expression ); // C++17
```

Компилятор проверяет static_assert объявление на наличие синтаксических ошибок при обнаружении объявления. Компилятор вычисляет параметр константного выражения немедленно, если он не зависит от параметра шаблона. В противном случае компилятор вычисляет параметр константного выражения при создании экземпляра шаблона. Таким образом, компилятор может вывести одно диагностическое сообщение, когда встретит объявление, а второе — когда будет создавать экземпляр шаблона.

Пример с областью видимости класса:

```
template <class T>
void swap(T& a, T& b) noexcept
{
    static_assert(std::is_copy_constructible<T>::value,
                  "Swap requires copying");
    static_assert(std::is_nothrow_copy_constructible<T>::value
               && std::is_nothrow_copy_assignable<T>::value,
                  "Swap requires nothrow copy/assign");
    auto c = b;
    b = a;
    a = c;
}
```

Пример ситуации, когда static_assert неприменим.

В целом, static_assert проверяет ровно то, что можно отловить на этапе компиляции. Если вы пытаетесь поймать что-то, что происходит в runtime, согласно изложенному выше, может возникнуть CE.

```
int main() {
    int x;
    cin >> x;
    static_assert(x==0);
    return 0;
}
```

5.1. Защищенное наследование. Логика видимости и доступности родителей и прародителей, их членов при двухуровневом наследовании с различными модификаторами доступа. Логика запрета и разрешения обращений к членам родителей при использовании слова friend в разных местах с различными комбинациями приватного и защищенного наследования.

Защищенное наследование: Есть три типа наследования

1. public: получить доступ к родительской части можно из любого места программы
2. protected: получить доступ к родительской части можно только изнутри класса, из его наследников и друзей
3. private: получить доступ к родительской части можно только из тела класса и из его друзей

При многоуровневом наследовании подниматься (получать доступ) по public и protected наследованиям можно сколько угодно, но если у нас на пути встретилось private наследование, то этого предка и то что выше него мы не видим (если мы не непосредственный ребенок). Если мы ребенок private наследования, то мы можем получить доступ к родителю и подниматься дальше по public и protected (до следующего private).

Friends: Рассмотрим пример

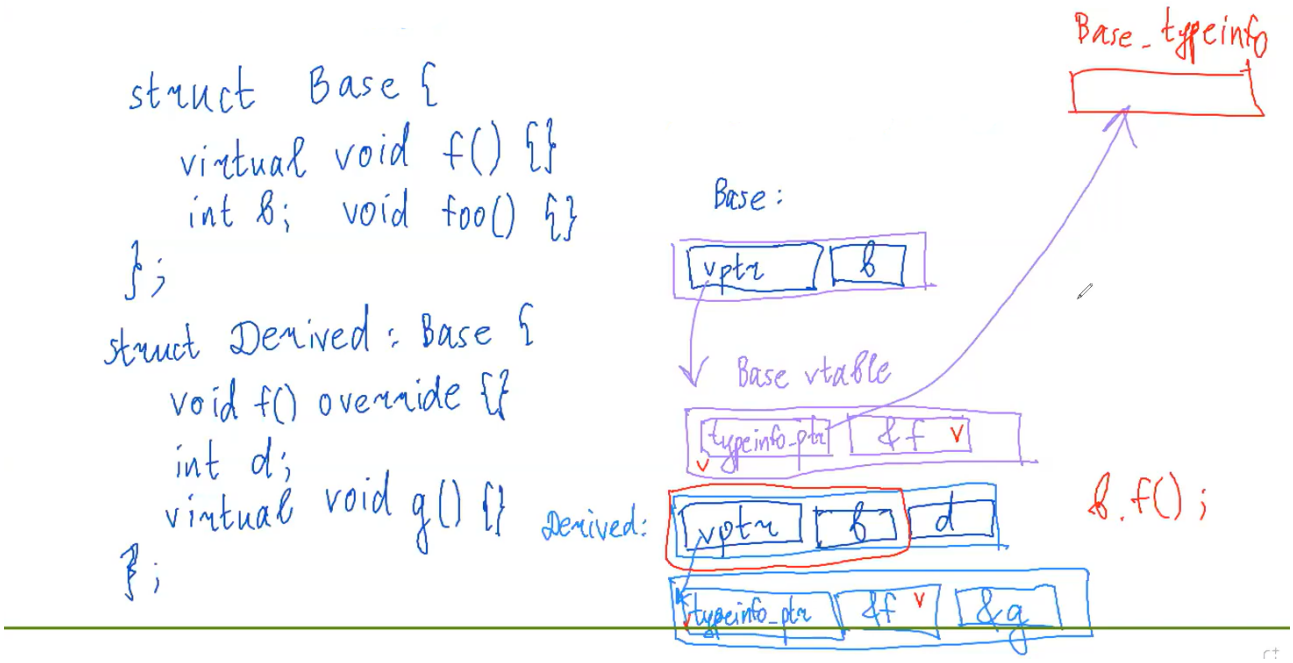
```
struct Grandma {          struct Mom: protected Grandma {
        int g = 0;                int m = 0;
};                        };

struct Daughter: private Mom {
    int d = 0;
    friend int main();
};

int main() {
    Daughter b;
    std::cout << b.g;
}
```

В этом случае все будет хорошо и g будет доступно из main (для protected в обоих местах аналогично). Однако если мы поменяем private и protected местами мы получим CE. Это интуитивно понятно: Grandma приватна для Mother, значит недоступна из ее наследников, а значит и из друзей наследников (очевидно что для двух private ситуация может только ухудшиться и доступа все еще не будет)

5.2. Понятие полиморфных объектов. Таблица виртуальных функций для полиморфного объекта, ее содержимое. Размещение в памяти объектов, у которых есть виртуальные функции. Объяснение, как за счет vtable происходит выбор нужной версии функции, а также за счет чего работают dynamic_cast и typeid. Разница между статическим и динамическим выбором версии функции с точки зрения реализации на низком уровне.

Рассмотрим полиморфный объект (у класса которого есть хотя бы один виртуальный метод) в памяти. Помимо его полей там будет храниться некоторый указатель - указатель на vtable. В этой таблице лежит указатель на место, где лежит typeinfo (нужно для typeid), а так же указатели на виртуальные функции.

Рассмотрим пример:



Как происходит выбор виртуальной f? В compile-time компилятор записывает инструкции: насколько шагов надо сдвинутся, чтобы получить указатель на f, и просто выполняет это в run-time.

Так как мы храним указатель на typeinfo работа typeid становится тривиальной. dynamic_cast теперь тоже может понять настоящий тип того что там лежит и сделать то что от него требуется (в примере dynamic_cast Base (красный прямоугольничек) к Derived (весь синий прямоугольничек)).

Заметим, что функция foo нигде не хранится рядом с объектами. Это объясняется тем, что про нее все известно в compile-time и нам не нужно ничего хранить в run-time для ее вызова.

Этим объясняется также разница статическим и динамическим выбором версии функции: статический - происходит в compile-time, динамический - в run-time через прыжки по нескольким указателям (что естественно медленнее чем в статическом случае).

5.3. Виртуальное наследование как решение проблемы ромбовидного наследования. Особенности размещения объектов в памяти при виртуальном наследовании. Почему при наличии виртуальных предков размер объекта увеличивается? Поведение компилятора в случае комбинации виртуального и обычного наследования одного и того же предка. Особенности видимости и доступности родительских методов в этих случаях.

Чтобы решить проблему ромбовидного наследования воспользуемся виртуальным наследованием. Оно создает только одну бабушку.

```
struct Granny {
    int g;
};

struct Mother: public virtual Granny {
    int m;
};

struct Father: public virtual Granny {
    int f;
};

struct Son: Mother, Father {
    int s;
};
```

Посмотрим на то как объект Son лежит в памяти

1. Без виртуального наследования: [g][m][g][f][s] (20 байт)
   Mother Father
2. С виртуальным наследованием: [m_ptr][m]....[f_ptr][f][s][g].... (40 байт)

   m_ptr, f_ptr - указатели на некоторое место в памяти, где лежит список указателей на виртуальных родителей (то есть на бабушку). Четыре точки - это пропуск в четыре байта, который возникает из-за выравнивания (адрес 8 байтовых чисел должен быть кратен восьми)

Из рассуждений выше видно, почему размер объекта увеличился (несмотря на то, что одна 4-байтовая бабушка исчезла добавилось 2 восьмибайтовых указателя)

Уберем у одного из родителей виртуальное наследование (допустим у Father). Теперь Son лежит в памяти так: [m_ptr][m][g][f][s][g] (одно g от папиной невиртуальной бабушки и одно от маминой виртуальной). Таким образом, проблему ромбовидного наследования таким способом не решить: обращение к g будет неоднозначным.

Замечание: В данных примерах не важно какое наследование (public/protected/private), так как проверка доступа происходит после выбора того к чему обращаемся, то есть неоднозначность никуда не денется.

5.4. Шаблонные шаблонные параметры, синтаксис использования. Пример: класс Stack на основе шаблонного контейнера. Шаблоны с переменным количеством аргументов (variadic templates). Синтаксис использования, пример: функция print. Пакеты аргументов и пакеты параметров, их распаковка. Реализация структуры is_homogeneous. Оператор “sizeof...”.

Шаблонные шаблонные параметры: В качестве параметров шаблона можно передавать другие шаблоны. Ниже приведен синтаксис использования на примере класса Stack.

```
template <typename T, template <typename U, typename Alloc> class Container
    = std::vector>
class Stack {
    Container<T, std::allocator<T>> c;
};

int main() {
    Stack<int, std::vector> s;
}
```

Variadic templates: Так же можно создавать шаблоны с переменным количеством параметров. Рассмотрим синтаксис на примере функции print от произвольного количества аргументов.

```
void print () {};

template <typename Head, typename... Tail> // Tail - пакет типов
void print(const Head& head, const Tail&... tail) { // tail - пакет аргументов
    std::cout << head << ' ';
    print(tail...); // многоточие распаковывает пакет (нужно для передачи в функцию)
}
```

Компилятор сгенерирует функции print от всех возможных наборов аргументов, от которых она вызывалась в программе. Возникает вопрос: зачем объявлять функцию print без аргументов? Почему нельзя было просто сделать так?

```
if (sizeof...(tail) == 0) { // оператор sizeof... возвращает
    print(tail...);         // количество элементов в пакете tail
}
```

Ответ прост: да, функция без аргументов в этом случае не вызовется, но она обязана будет скомпилироваться. Если не объявить ее получим CE.

Метафункция `is_homogeneous` - обобщение `is_same` на произвольное количество аргументов:

```
template <typename First, typename Second, typename... Types>
struct is_homogeneous {
    static const bool value = is_same_v<First, Second> &&
                              is_homogeneous<Second, Types...>::value;
};

template <typename First, typename Second>
struct is_homogeneous<First, Second> {
    static const bool value = is_same_v<First, Second>;
};
```

5.5. Зависимые имена. Пример неоднозначности между declaration'ом и expression'ом. Применение ключевого слова typename для устранения неоднозначности с зависимым именем. Применение слова template для решения аналогичной проблемы с зависимыми именами шаблонов (иллюстрация примером).

Рассмотрим такой пример

```
template <typename T>          template <>
struct S {                     struct S<int> {
    using X = T;                   static int X;
};                             };

template <typename T>
void f() {
    S<T>::X* a;
}
```

Заметим, что строчку из функции f можно трактовать двояко

1. declaration: объявляем переменную a типа `S<T>::X*`
2. expression: умножаем переменную `S<T>::X` на переменную a

В данном случае X является зависимым именем, то есть его смысл зависит от шаблонного параметра T (может даже не зависеть в данный момент, но потенциально зависеть). По умолчанию компилятор читает такие случаи как expression. Чтобы исправить это, необходимо написать `typename S<T>::X* a`. Теперь эта строчка будет читаться как объявление переменной a.

Замечание: Неважно какой знак там стоит (хоть никакого), важно что компилятор пытается трактовать все зависимые имена как переменные и не может скомпилировать эту строку без слова typename.

Рассмотрим другой пример

```
template <typename T>              template <>
struct SS {                        struct SS<int> {
    template <int M, int N>            static const int A = 0;
    struct A {};                   };
};

template <typename T>
void g() {
    SS<T>::A<1, 2> a;
}
```

И снова получаем неоднозначность

1. Объявляем переменную a типа `S<T>::A<1,2>`
2. `(SS<T>::A < 1), (2 > a)`

Одного слова typename нам тут уже не хватит, так как когда компилятор видит слово typename он ждет название типа, а в данном случае он получает название шаблона. Эта проблема решена с помощью слова template: `typename<T>::template A<1, 2> a;`

Замечание: Иногда нам может понадобиться слово template, но не понадобиться typename, например при использовании шаблонных функций и т. д. (то есть не классов)

5.6. Реализуйте класс std::insert_iterator и функцию std::inserter. Объясните, как работает ваша реализация. Приведите пример использования этого класса и этой функции.

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()));
```

Реализации std::insert_iterator и std::inserter

```
#include <iterator>

template <typename Container>
class insert_iterator {
 public:
  // необходимые для работы с std::iterator_traits typedef-ы (помеченные как void считаются
    невалидными)
  // (подробности см. в реализации reverse_iterator)
  typedef void value_type;
  typedef void reference;
  typedef void difference_type;
  typedef void pointer;
  typedef std::output_iterator_tag iterator_category;

  explicit insert_iterator(Container& container, typename Container::
    iterator it) : container(&container), iter(it) {} //
    (1)
  // инкремент ничего не делает
  insert_iterator& operator++()    noexcept { return *this; };
  insert_iterator& operator++(int) noexcept { return *this; };
```

```
  // ничего не делает, но возвращает insert_iterator чтобы можно было писать *it = value
  insert_iterator& operator*()      noexcept { return *this; };

  // не операторы копирования/перемещения, а операторы присваивания элементам контейнера!
  // у контейнера должен быть определён typedef: value_type - тип элементов контейнера
  insert_iterator& operator=(const typename Container::value_type& value) {
    // iter необходимо обновлять результатом операции, т.к. он может поменяться после
   реаллокации
    iter = container->insert(iter, value);
    ++iter;
    return *this;
  }
  // вторая версия для вставки rvalue элементов
  insert_iterator& operator=(typename Container::value_type&& value) {
    iter = container->insert(iter, std::move(value));
    ++iter;
    return *this;
  }

  // наследники могут хотеть иметь доступ к контейнеру и итератору, поэтому не private
 protected:
  // проще хранить указатель, чем возиться с копиями и ссылками
  Container* container;
  typename Container::iterator iter;
};

template <typename Container>
insert_iterator<Container> inserter(Container& container, typename Container
   ::iterator it) {
  return insert_iterator<Container>(container, it);
}

/* Примечания:
(1) Итераторы принято передавать по значению, тк.. они достаточно малы, и ссылки не
   дают преимущества перед передачейпо
 значению. */
```

5.7. Реализуйте классы std::istream_iterator и std::ostream_iterator. Объясните, как работает ваша реализация. Приведите пример использования этих классов.

Задача: Считать из файла input.txt массив целых чисел, разделенных пробельными символами. Отсортировать их и записать в файл output.txt

Решение:

```
#include <vector>
#include <algorithm>
#include <fstream>

int main(){
  // открываем input.txt для чтения
  std::ifstream fin("input.txt");
  // открываем output.txt для записи
  std::ofstream fout("output.txt");
  // объявление и инициализация пустого целочисленного вектора
  std::vector<int> v;

```

```
  // сложная магия, благодаря которой из потока чтения вставляются элементы в конец вектора
  std::copy(std::istream_iterator<int>(fin), std::istream_iterator<int>(),
   std::inserter(v, v.end()));
  // алгоритм сортировки
  std::sort(v.begin(), v.end());
  // сложная магия, благодаря которой элементы из вектора копируются в поток записи
  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(fout, " "));
  return 0;
}
```

- Одной из основ библиотеки являются итераторы, а также полуинтервалы, ими определяемые. По семантике (читай — по поведению) они совпадают с указателями. То есть, опреатор разыменования * вернет вам элемент, на который ссылается итератор, ++ переведет итератор на следующий элемент. В частности, любой контейнер представляется его концевыми итераторами [begin, end), где begin указывает на первый элемент, end — за последний;

- Алгоритмы, работающие с контейнерами, в качестве параметров принимают начало и конец контейнера (или его части);

- Алгоритм копирования copy просто переписывает элементы из одного полуинтервала в другой. Если в целевом контейнере не выделена память, то поведение непредсказуемо [copy];

- Функция `inserter` вставляет значение в контейнер перед итератором [`inserter`]

- `istream_iterator` и `ostream_iterator` предоставляют доступ к потокам в стиле контейнеров [`istream_iterator, ostream_iterator`]

Ещё один пример использования:

```
#include <iostream>
#include <sstream>
#include <iterator>
#include <numeric>
#include <algorithm>

int main()
{
    std::istringstream str("0.1 0.2 0.3 0.4");
    std::partial_sum(std::istream_iterator<double>(str),
                     std::istream_iterator<double>(),
                     std::ostream_iterator<double>(std::cout, " "));

    std::istringstream str2("1 3 5 7 8 9 10");
    std::cout << "\nThe first even number is " <<
        *std::find_if(std::istream_iterator<int>(str2),
                      std::istream_iterator<int>(),
                      [](int i){return i%2 == 0;})
        << ".\n";
    //"9 10"left in the stream
    // Вывод:
    // 0.1 0.3 0.6 1
    // The first even number is 8.
}
```

Реализация:

```
    template <typename T>
    class istream_iterator {
        std::istream& in;
        T value;
    public:
        istream_iterator(std::istream& in): in(in) {
```

```
            in >> value;
        }
        istream_iterator<T>& operator++() {
            in >> value;
        }
        T& operator*(){
            return value;
        }
    };
    // children:
    // std::ifstream in("input.txt");
    // std::istringstream iss(s);
```

5.8. Особенности перегрузки операторов new и delete. Разница между оператором new и функцией operator new. Разновидности оператора new, placement new, nothrow new, их правильная перегрузка. Особенности вызова нестандартного operator delete. В каком случае компилятор способен вызвать нестандартный operator delete самостоятельно?

Оператор **new** помимо выделения памяти под какой-то тип, еще и направляет конструкторы для него в каждую ячейку выделенной памяти. Этим оператор new отличается от сишного malloc (та только выделяет столько байт сколько попросили, конструкторы элементов приходилось вызывать вручную).

Действия оператора new можно перегрузить, но не целиком - только ту часть, которая отвечает выделению памяти. Конструкторы будут неизбежно вызваны после выделения памяти.

Отсюда следует **разница между оператор new и функцией оператор new**: при создании нового объекта память выделяется с помощью функции operator new, а затем вызывается конструктор для инициализации памяти. Здесь оператор new выполняет и выделение, и инициализацию, в то время как функция operator new выполняет только выделение.

Пример: у структуры S конструктор сделан приватным, тогда new S (или new S()) не скомпилируется, а operator new(sizeof(S)) (но нужно сделать reinterpret_cast от void* к S*, но вообще-то так делать не стоит) скомпилируется, и выделит память с помощью глобального new, без вызова конструктора

Заметка: для operator delete reinterpret_cast можно не делать

По станадрту оператор new принимает число - сколько нужно выделить байт

Оператор delete сначала вызовет деструкторы, потом очистит память, перегрузке аналогично подлежит только очищение памяти

Выделяем память с помощью сишной функции malloc(n) - запрашивает у ядра операционной системы память в байтах и возвращает указатель на выделенную память, если не получилось то nullptr.

Оператор **delete** принрмает в качестве аргумента указатель. Си-шная функция освобождает память - free.

Операторы new и delete для массивов отличаются от обычных

```
1   void* operator new(size_t n){
2       // return malloc(n);
3       void p* = malloc(n);
4       if (!p) throw std::bad_alloc();
5       return p;
6   }
7   void* operator new[](size_t n){
8       void p* = malloc(n);
9       if (!p) throw std::bad_alloc();
10      return p;
11  }
12  void operator delete(void* ptr){
13      free(ptr)
14  }
15  void operator delete[](void* ptr){
16      free(ptr)
```

```
    }
```

Если память не удалось выделить, то в стандартой реализации new вызывается функция **new_handler**, которая может другим способом попробовать выделить память (возможно с диска). Кроме того, можно самим переопределить функцию new_handler с помощью **set_new_handler**. После того как он ее вызывает, он снова пытается сделать malloc. Если попросим выделить 0 байт, оператор new выделит все равно 1 байт, так как в Си могут быть указатели на один и тот же адрес, а в C++ это запрещено.

Можно перегрузить оператор new для конркретного типа - для этого пишем оператор new в теле класса. Так как перегрузка оператора общая для всех объектов класса, то функция перегрузки оператора должна быть static. (Правило “частное лучше общего” при выборе new так же актуально); P.S. это можно не писать и оно и так будет считаться статик А еще можно делать операторы с кастомными параметрами - тогда при вызове new с такими параметрами, будет вызываться перегруженный оператор, но стандартный при этом так же будет работать.

Placement new

Напоминание: Если выделен какой-то кусок памяти под S, но конструктор на нем еще не было вызван, то есть синтаксис чтобы направить конструктор на уже выделенную память по указателю

```
    S* p = reinterpret_cast<S*>(operator new(sizeof(S)));
    S* p1 = reinterpret_cast<S*>(operator new(sizeof(S)));
    new(p) S(); //default construct will be called
    new (p1) S(value); //construct from value
```

Заметка: если в структуре переопределен оператор new, то placement new для нее работать не будет

Можно перегрузить оператор **new** именно для **placement new**. По умолчанию оператору new передается сколько памяти нужно выделить (компилятор подставлят это сам в обычном new), однако здесь память уже выделена, поэтому мы не пользуемся этим. Кроме того, как уже говорилось выше, в операторе new можно перегрузить только часть с выделением памяти, но в нашем случае память уже выделена, поэтому данный оператор ничего по сути не делает.

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

No-throw оператор new - не бросающий оператор new; он возвращает nullptr, если не удалось выделить память.

```
    // Перегрузка:
    void* operator new(size_t n, std::nothrow_t) {
        return malloc(n);
    }

    // Пример использования:
    int main() {
        int* ptr = new(std::nothrow) int(5);
    }
```

Placement delete не существует. Если хотим вызывать delete с кастомными параметрами, нужно будет вызывать функцию оператор delete явно: operator delete(ptr, mystruct).В таком случае нужно будет еще отдельно вызвать деструктор.

```
    S* ptr = new(mystruct) S();
    operator delete(ptr, mystruct);
    p->~S()
```

На самом деле компилятор иногда умеет вызывать кастомный оператор delete самостоятельно - только в случае если конструктор бросил исключение, так как компилятор должен подчистить выделенную память (сам вызывает кастомный delete) если он может это сделать (если кастомного delete нет - то ничего не происходит).

5.9. Реализуйте стековый аллокатор. Это такой аллокатор, который заводит большой массив в стековой памяти и всю память берет из него, ни разу не обращаясь к new (тем самым давая выигрыш во времени для контейнера, построенного на нем). Он делает это в предположении, что количество запрошенной памяти никогда не превзойдет размер этого массива.

PoolAllocator/StackAllocator - когда аллокатор конструируется в нем выделятся сразу большой пул огромного размера, а дальше при его allocate этот аллокатор хранит в себе одно число - указатель на первый незанятый байт (кратный 4 или 8), сдвигает этот указатель на соответсвующее число и вовзрвщает кусочек на котором он стоял до этого. При deallocate он не делает ничего. При вызове деструктора, удаляется весь пул

Когда это нужно: когда знаем что памяти много и ее заведомо хватит чтобы весь контейнер поместится. Такой аллокатор дает существенный выигрыш по времени в тех контейнерах, в которых каждое добавление - это вызов new(list, map, unordered_map)

Вообще, при таком аллокаторе даже не всегда приходится выделять динамическую память. Просто создает на стеке создает массив, а дальше ведет себя как этот аллокатор (если не больше 100 тысяч элементов, то такое сработает)

```
#pragma once
#include <vector>
#include <ctime>

#define tchS template<size_t chunkSize>
#define tT template<typename T>
#define tU template<typename U>
#define tTU template<typename T, typename U>
const int kAllocSz = 1000;

tchS
class FixedAllocator {
 private:
  std::vector<void*> pool_; // храним вектор свободных значений
 public:
  FixedAllocator() = default;
  ~FixedAllocator() = default;

  static FixedAllocator<chunkSize>& get_alloc() {
    static FixedAllocator<chunkSize> a;
    return a;
  }

  void* allocate();
  void deallocate(void* el, size_t t) { if (t == chunkSize) pool_.push_back(
   el); }
```

```
};


// ==================== Реализация FixedAllocator ==============================
tchS
void* FixedAllocator<chunkSize>::allocate() {
  if (pool_.size() == 0) {
    try {
      char* ptr = static_cast<char*>(::operator new(kAllocSz * chunkSize));
      for (int i = 0; i < kAllocSz; ++i)
        pool_.push_back(static_cast<void*>(ptr + i * chunkSize));
    } catch (...) {
      throw;
    }
  }
  void* ptr = pool_.back();
  pool_.pop_back();
  return ptr;
}
```

5.10. Понятие allocator-aware контейнеров. Параметры аллокаторов propagate_on_container_copy_assignment /move_assignment / swap, их предназначение и использование. Функция select_on_container_copy_construction, ее предназначение и использование. Реализация allocator-awareness на примере контейнера vector: правильная работа с аллокатором при копировании/перемещении/присваивании контейнера.

AllocatorAwareContainer - это контейнер, который содержит экземпляр аллокатора и использует этот экземпляр во всех своих функциях-членах для выделения и деаллокации памяти, а также для создания и уничтожения объектов в этой памяти (такими объектами могут быть элементы контейнера, узлы или, для неупорядоченных контейнеров, массивы ведер).

The following rules apply to container construction

- Copy constructors of *AllocatorAwareContainer*s obtain their instances of the allocator by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator of the container being copied.
- Move constructors obtain their instances of allocators by move-constructing from the allocator belonging to the old container.
- All other constructors take a `const allocator_type&` parameter.

The only way to replace an allocator is copy-assignment, move-assignment, and swap:

- Copy-assignment will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value` is `true`
- Move-assignment will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment::value` is `true`
- Swap will replace the allocator only if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`. Specifically, it will exchange the allocator instances through an unqualified call to the non-member function swap, see *Swappable*.

Что вообще такое **копирование аллокатора** (что значит инициализировать один аллокатор другим аллокатором)?

Допустим у нас PoolAllocator. Если мы копируем аллокатор, то пул копировать не надо,

так как новый аллокатор должен уметь освобождать то, что выделил старый аллокатор.

```
alloc1 == alloc2
// this means
T* ptr = alloc1.allocate(1);
alloc2.deallocate(ptr, 1);
```

Чтобы несколько аллокаторов могли указывать на один и тот же пуул нужно использовать **shared_ptr<Pool>**, который принимает обычный C-style поинтер. И в деструкторе мы удаляем пуул, только тогда, когда наш указатель на пуул последний.

Поведение аллокатора при копировании и присваивании контейнера

Нужно ли нам в таком случае делать копию аллокатора или необходимо создать новый.

```
vector<int, PoolAlloc> v1;
// .... //
vector<int, PoolAlloc> v2 = v1;
```

1 вариант: мы хотим чтобы копия контейнера указывала на старый пулл

2 вариант: мы хотим чтобы у каждого контейнера был свой пуул.

В какой момент принимается решение копировать/не копировать аллокатор? Для этого в allocator_traits есть специальный метод **select_on_container_copy_construction**. Этот метод возвращает объект аллокатора, который будет использоваться в контейнере. По умолчанию вернётся копия аллокатора (два контейнера будут указвать на один и тот же пуул).

Что должен делать контейнер при копировании. Мы должны инициализировать аллокатор результатом выше упомянутого метода.

Также в allocator_traits есть using **propagate_on_container_copy_assignment**, который определяет нужно ли заниматься присваиванием аллокатора при присваивании контейнера. По умолчанию он равен std::false_type (в нем static bool_value = false). Но можно сделать его true. Аналогичная история со swap - **propagate_on_container_swap**

Оператор присваивания в этой реализации не безопасен относительно исключений!

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
    T* arr;
    size_t sz, cap;
    Alloc alloc;

    using AllocTraits = std::allocator_traits<Alloc>;
public:
    Vector(size_t n, const T& val = T(), const Alloc& alloc = Alloc());

    Vector<T, Alloc>& operator=(const Vector<T, Alloc>& other) {
        if (this == &other) return *this;

        for (size_t i = 0; i < sz; ++i) {
            pop_back();
            //AllocTraits::destroy(alloc, arr + i);
        }
        AllocTraits::deallocate(arr, cap);

        //main decision
        if (AllocTraits::propagate_on_container_copy_assignment::value
&& alloc != other.alloc) {
            alloc = other.alloc;
            // what if the exception appear here??
        }

        sz = other.cap;
```

```
            cap = other.cap;

            AllocTraits::allocare(alloc, other.cap);
            for (size_t i = 0; i < sz; ++i) {
                push_back(other[i]);
            }
             return *this;
        }
    }
```

Отныне мы все чаще и чаще будем обращаться к разделу **named requirements** на cpp.reference

Например, наша реализация list должна быть написана согласно *AllocatorAwareContainer*.

```
\textbfПример{ select_on_container_copy_construction:}

template <typename T, typename Allocator>
List<T, Allocator>::List(const List<T, Allocator>& another):    allocator_(
    AllocatorTraits::select_on_container_copy_construction(another.allocator_
    )),
                                                                    head(
    AllocatorTraits::allocate(node_allocator_, 1)) {
    head->previous = head;
    head->next = head;
    for (auto it = another.begin(); it != another.end(); ++it)
        push_back(*it);
}
```

5.11 Реализация функции std::forward. Объяснение ее действия. Объяснение, почему принимаемый и возвращаемый типы именно такие. Почему в качестве принимаемого типа нельзя написать T&&?

Когда у нас в шаблонную функцию передается пременное число аргументов, и мы не знаем какие из них rvalue, а какие lvalue, можем воспользоваться функцией std::forward

```
template <typename ...Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Теперь все типы, кототые были переданы как lvalue будут иметь тип + &, а которые были переданы как rvalue - тип + &&. Пусть внутри функции f вызывается функция g, принимающая пакет аргументов, хотелось бы применить std::move к тем, которые являются rvalue и скорпировать остальные (к ним нельзя применять std::move, так как их передали в f не как rvalue, следовательно не ожидают удаления всей инфорации)

Реализация

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

Такая конструкция породит rvalue для объектов которые были изначальное отданы как rvalue и lvalue для всех остальных, это значит все аргументы проходят в функцию g с такими же видами value с какими нам их дали, как следствие сможем копировать только те объекты, которые нам изначально пришлось бы копировать. - **perfect forwarding** .

Именно perfect forwarding объясняет std::forward. Нам передаются аргументы, и мы не хотим постоянно тупо копировать (а в случае с rvalue создаются лишние копии). Тогда мы используем std::forward.

Полезная ссылка-пояснение. TL;DR - C++ 11 пытается сохранить rvalue как может.

5.12 Return Value Optimization (RVO), ее идея и пример, когда она возникает. Пример, когда небольшая модификация возвращаемого выражения приводит к исчезновению RVO (оператор += в BigInteger). Примеры, когда надо и когда не надо писать std::move после return. Понятие copy elision, идея этой оптимизации и пример ее возникновения. Условия, при которых она точно происходит, а также при которых может произойти, но не обязана.

Пускай мы перегружаем операторы.

Перегрузка арифметических операторов

В метод класса передаем только второй операнд, так как под левым операндом подразумевается *this.

```
struct Complex{
    double re = 0.0;
    double im = 0.0;

    Complex& operator +=(const Complex& z) {
        re += z.re;
        in += z.im;
        return *this;
    }
    Complex operator +(const Complex& z) {
        Complex copy = *this;
        copy += z;
        return copy;
    }

}

```

Оператор '+' должен создать копию объекта. Тогда если бы нам захотелось реализовать '+=' через '+' то на любое действие, даже при добавлении одного симовла к строке, создавалась бы ее копия и время работы оператора за счет копирования увеличивалось бы до $O(n)$, тогда как добавление одного символа к строке может быть реализовано за $O(1)$ - неэффективное решение.

Напоминание : возвращаем значение по ссылке, а не по указателю, так как ссылка ничего не весит.

Оператор '+' нужно определять вне класса, так как например для данной структуры при вызове

```
Complex c(2.0);
c += 1.0;
```

или при вызове

```
Complex c(2.0);
c + 1.0;
```

все сработает хорошо, компилятор сделает неявное преобразование double к Complex, однако следующий вызов

```
Complex c(2.0);
1.0 + c;
```

выдаст ошибку, так как мы определили оператор '+' только тогда, когда левым операндом является объект класса (*this).

При определении оператора вне функции и левый, и правый операнд будут равноправны, и компилятор сможет делать каст как левого, так и правого операнда - соответственно в оператор надо передавать два параметра. Тогда корректный код выглядит так:

```
struct Complex{
    double re = 0.0;
    double im = 0.0;

    Complex (const Complex&){
    }

    Complex& operator +=(const Complex& z) {
        re += z.re;
        in += z.im;
        return *this;
    }
}
```

```
14
15      Complex operator +(const Complex& a, const Complex& b) {
16              Complex copy = a; //Copy constructor definitely called
17              return copy += b;
18              //copy += b;
19              //return copy;
20      }
21
22      int main(){
23          Complex c(2.0);
24          Complex d (1.0,3.0);
25          Complex sum = c + d; //Copy constructor isn't called
26      }
27
```

В данном коде конструктор копирования вызовется 2 раза: в (16) строке - при создании copy, и в (17) - при возвращении результата (метод возвращает результат по значению, а значит создается копия результата). В (25) строке при присваивании суммы конструктор копирования не вызывается - происходит **copy elision** (появилось в C++11). Copy elision заключается в следующем: справа от оператора '=' после выполнения операции создался временный объект типа Complex, которым инициализируется левый операнд. Тогда компилятор не создается еще один временный объект для присваивания, а сразу считает получившийся временный объект нужным. Можно сократить количество копирований до одного с помощью **Return Value Optimization(RVO)** - если компилятор понимает, что в методе создается локальный объект и он же возвращается, то компилятор выделит память в том месте, где ожидается возвращание результата функции, таким образом убирая лишнее копирование. Заметим, что в незакомментированном коде это оптимизация не будет вызвана, хотя и возвращается объект, созданный в методе - компилятору не очевидно, что это тот же самый объект.

Очень полезная ссылка

Про std::move и return:

Иногда хочется написать std::move после return, чтобы соптимизировать, но так делать не всегда надо; из-за RVO это может ухудшить ситуацию. Например:

если возвращается не ссылочный тип, то из-за RVO компилятор может избежать перемещение И копирование, но std::move заставит делать перемещение, нет оптимизации. Поэтому если есть RVO, писать return std::move не надо, например:

Если вы имеете дело с функцией, осуществляющей возврат по значению, и возвращаете объект, привязанный к rvalue-ссылке или универсальной ссылке, вы захотите применять std::move или std::forward при возврате ссылки. Чтобы понять, почему, рассмотрим функцию operator+ для сложения двух матриц, где о левой матрице точно известно, что она является rvalue (а следовательно, может повторно использовать свою память для хранения суммы матриц):

```
1 Matrix operator+ ( matrix&& lhs , const Matrix& rhs ) // Возврат по значению
2 {
3     lhs += rhs ;
4     return std:: move(lhs); // Перемещение lhs в
5 } // возвращаемое значение
```

Подробнее про RVO

5.13. Контейнер deque: внутреннее устройство с алгоритмической точки зрения, сходства и различия с vector. Что хранится внутри deque, как происходит его расширение? Объяснение асимптотики работы методов push_back/push_front и [ ], за счет чего она достигается. Как устроены итераторы в deque, что хранят итераторы? Разница между инвалидацией итераторов и инвалидацией ссылок, правила инвалидации итераторов и ссылок в deque с объяснением, почему они такие.

Посмотрим на контейнеры:

Container	indexating[]	push_front	push_back	insert(it)	erase(it)	find	iter
vector	$O(1)$	-	$O(1)$ amort	$O(n)$	$O(n)$	-	RA
deque	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	-	RA

Deque реализован как массив массивов (T**). В каждой ячейке большого массива есть указатель на элементы другого массива, где уже лежат элементы. Также храним номер ячейки (номер "строки"и номер "столбца") начала + номер ячейки, где конец дека. Всё промежуточное пространство заполнено элементами. [ ] по индексу - вычисляем исходя из позиции начала и размера одного массива, куда достать. Итератор random access - так как он хранит вот эти два индекса (не просто указатель) ⇒ за O(1) можно пересчитать в двумерном массиве прибавление int. Push_front и push_back работают за O(1), см. билет Поли.

Инвалидация итераторов:

Iterator invalidation

This section is incomplete

There are still a few inaccuracies in this section, refer to individual member function pages for more detail

Operations	Invalidated
All read only operations	Never
swap, std::swap	The past-the-end iterator may be invalidated (implementation defined)
shrink_to_fit, clear, insert, emplace, push_front, push_back, emplace_front, emplace_back	Always
erase	If erasing at begin - only erased elements If erasing at end - only erased elements and the past-the-end iterator Otherwise - all iterators are invalidated (including the past-the-end iterator).
resize	If the new size is smaller than the old one : only erased elements and the past-the-end iterator If the new size is bigger than the old one : all iterators are invalidated Otherwise - none iterators are invalidated.
pop_front	Only to the element erased
pop_back	Only to the element erased and the past-the-end iterator

Invalidation notes

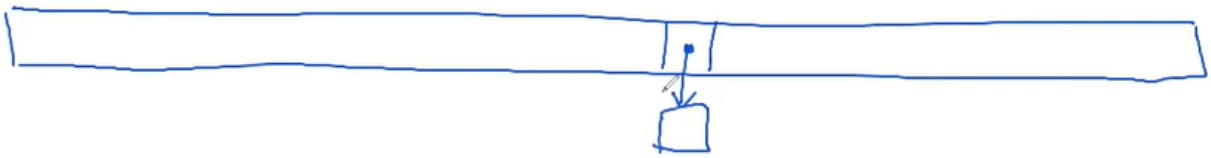
- When inserting at either end of the deque, references are not invalidated by insert and emplace.
- push_front, push_back, emplace_front and emplace_back do not invalidate any references to elements of the deque.
- When erasing at either end of the deque, references to non-erased elements are not invalidated by erase, pop_front and pop_back.
- A call to resize with a smaller size does not invalidate any references to non-erased elements.
- A call to resize with a bigger size does not invalidate any references to elements of the deque.

5.14 Контейнеры unordered_map и unordered_set, их внутреннее устройство с алгоритмической точки зрения: что хранится внутри, какие алгоритмы и структуры данных используются для реализации методов. Что алгоритмически происходит при вставке, при удалении, при rehash? Пример использования нестандартного компаратора, нестандартной хэш-функции в unordered_map. Асимптотика обхода unordered_map итератором с объяснением, как он (обход) работает. Правила инвалидации итераторов и ссылок в unordered_map.

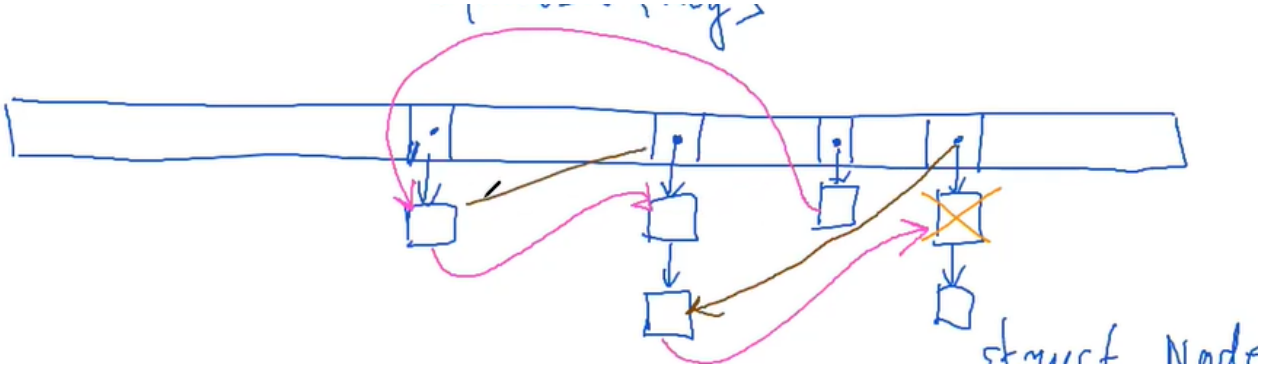
unordered_map https://en.cppreference.com/w/cpp/container/unordered_map

```
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> >
> class unordered_map;
```

unordered_map - это хеш-таблица, которая позволяет выполнять операции за O(1) в среднем за счет потери упорядоченности. Изначально эта таблица пустая (заполнена nullptr), в конкретный bucket попадают элементы, чей хеш равен номеру bucket



Unordered_map представляет собой некоторое количество односвязных списков, конец каждого из которых указывает на начало следующего, а начало каждого списка - на конец предыдущего



insert

Если нужно положить элемент в ячейку, в ячейке заводится односвязный список, этот односвязный список вставляется в начало таблицы и в него вставляется элемент, после чего добавляем

указатель из конца нового списка в начало следующего. Если там уже был не пустой список при вставке, то нужно пройтись по всему списку и с помощью переданного компаратора equal_to сравнить ключи. Так как в unordered_map не может быть двух элементов с одинаковым ключом, мы либо обновляем элемент в таблице (если ключ нашелся), либо добавляем новый в конец списка, обновляем связи.

erase

Чтобы удалить элемент, мы аналогично insert его ищем, и в случае, если элемент действительно лежит в контейнере, нам нужно удалить его из списка и обновить указатели на конец предыдущего списка/начало следующего списка в случае, если был удален соответственно первый или последний элементы односвязного списка в bucket

rehash

При rehash мы создаем таблицу большего размера, перекладываем все элементы старой таблицы в новую и удаляем старую

Обходя unordered_map по итератору, мы просто итерируемся по одному длинному списку, поэтому асимптотика O(n)

В качестве параметра Hash и KeyEqual можно передать свои нестандартные хеш-функцию и компаратор

```cpp
template<typename T>
struct MyHash
{
    std::size_t operator()(T const& value) const
    {
        size_t h1 = std::hash<T>{}(value);
        size_t h2 = std::hash<T>{}(value);
        return h1 ^ (h2 << 1);
    }
};
```

```cpp
template<typename T>
struct MyEqual
{
    std::size_t operator()(T const& value1, T const& value2) const
    {
        return (value1 > (value2 + value2));
    }
};
```

Правила инвалидации итераторов и ссылок в unordered_map

unordered_map **не инвалидирует ссылки**, а **итераторы** инвалидируются **только если произошел rehash**

unordered_set

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.
https://en.cppreference.com/w/cpp/container/unordered_set

5.15 Гарантии безопасности контейнеров относительно исключений. Проблемы с гарантиями exception-safety у map и unordered_map. Реализация exception safety на примере vector (с произвольным аллокатором): реализация безопасных конструкторов, операторов присваивания, метода push_back.

Гарантии безопасности относительно исключений

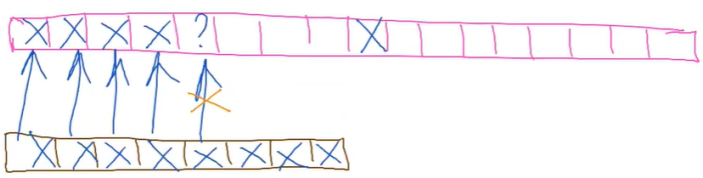
Рассмотрим пример работы вектора

```
void f() {
    Dangerous s(0);
    std::cout << s.x;
    g();
}

struct S {
    int x = 0;
    S(int x): x(x) {}
    S(const S& s): x(s.x) {
        if (x == 8)
            throw 1;
    }
};

int main() {

    std::vector<S> v;
    for (int i = 0; i < 100; ++i) {
        v.push_back(S(i));
    }


    try {
        f();
    } catch (...) {
        std::cout << "caught\n";
    }
}
```

Как мы уже знаем, при заполнении вектора полностью. он расширяется в два раза. Давайте представим, что мы положили какой-то объект, который при копировании бросаем исключение. Чт делать в этот момент вектору? Тут две возможности

1 Внетор может столкнуться с этим исключением, когда он кладет в себя объект

2 А может быть еще хуже. На восьмом шаге у него будет происходить следующее: он начнет перекладывать все преедыдущие объекты в новое место, и если в этот момент кто-то из тех, кого он перекладывает, выброст исключение, случится беда. Он уже часть нового storage выделил, поэтому если вектор реализован плохо, контейнер окажется в разломанном состоянии. Чтобы такого не было, нужно уничтожить все элементы, которые мы успели переложить, освободить память и вернуть все, как было



Функции могут давать или не давать гарантию безопасности относительно исключений. Контейнеры могут перестать работать корректно вследствие вызова исключений и вызвать UB.

Базовая (basic) гарантия: объект останется в валидном состоянии после вызова исключений

Сильная (strong) гарантия: объект останется в исходном состоянии после выхова исключений.

Почти все STL-библиотечные функции дают сильную гарантию безопасности.(vecor дает такую гарантию. Он устроен так, что если во время push_back вылетело исключение, он все вернет аккуратно, как было, и ничего не сломает)

Vector

```
void reserve(size_t n) {
    if (n <= cap) return;

    //T* newarr = alloc.allocate(n); // WHY??
    T* newarr = AllocTraits::allocate(Alloc, n);

    size_t i = 0;
    try {
        for (; i < sz; ++i) {
            //AllocTraits::construct(alloc, newarr + i, arr[i]);
            AllocTraits::construct(alloc, newarr + i, std::move(arr[i]));
        }
    } catch(...) {
        for (size_t j = 0; j < i; ++j){
            AllocTraits::destroy(alloc, newarr + j);
        }
        AllocTraits::deallocate(newarr, n);
        throw;
    }

    for (size_t i = 0; i < sz; ++i) {
        AllocTraits::destroy(alloc, arr + i);
    }
    AllocTraits::deallocate(arr, n);
    arr = newarr;
    cap = n;
}
```

```
Vector (const Vector& other){
    reserve(other.cap);
    sz = other.sz;

    size_t i = 0;
    try {
        for (; i < sz; ++i) {
            AllocTraits::construct(alloc, arr + i, other.arr[i]);
        }
    } catch(...) {
        for (size_t j = 0; j < i; ++j){            AllocTraits::destroy(alloc,
   arr + j);
        }
        AllocTraits::deallocate(arr, cap);
        throw;
    }
}
```

```
Vector (Vector&& other){
    reserve(other.cap);
    sz = other.sz;

    other.cap = 0;
    other.sz = 0;

    size_t i = 0;
    try {
        for (; i < sz; ++i) {
            AllocTraits::construct(alloc, arr + i, std::move(other.arr[i]));
        }
    } catch(...) {
        for (size_t j = 0; j < i; ++j){
            AllocTraits::destroy(alloc, arr + j);
        }
        AllocTraits::deallocate(arr, cap);
        throw;
    }
}
```

```
Vector& operator = (const Vector& other){

    if(this != addressof(other)){
        reserve(other.cap);
        sz = other.sz;

        size_t i = 0;
        try {
            for (; i < sz; ++i) {
                AllocTraits::construct(alloc, arr + i, other.arr[i]);
            }
        } catch(...) {
            for (size_t j = 0; j < i; ++j)
                AllocTraits::destroy(alloc, arr + j);
            AllocTraits::deallocate(arr, cap);
            throw;
        }
    }
    return *this;
}
```

```
Vector& operator = (Vector&& other){

    if(this != addressof(other)){
        reserve(other.cap);
        sz = other.sz;

        other.cap = 0;
        other.sz = 0;

        size_t i = 0;
        try {
            for (; i < sz; ++i) {
                AllocTraits::construct(alloc, arr + i, std::move(other.arr[i]));
            }
        } catch(...) {
            for (size_t j = 0; j < i; ++j)
                AllocTraits::destroy(alloc, arr + j);
            AllocTraits::deallocate(arr, cap);
            throw;
        }
    }
    return *this;
}
```

```
    void push_back(const T& value) {
        if (sz == cap)
            reserve(2 * cap);
        //new(arr + sz) T(value);
        AllocTraits::construct(alloc, arr + sz, value);
        ++sz;
    }
```

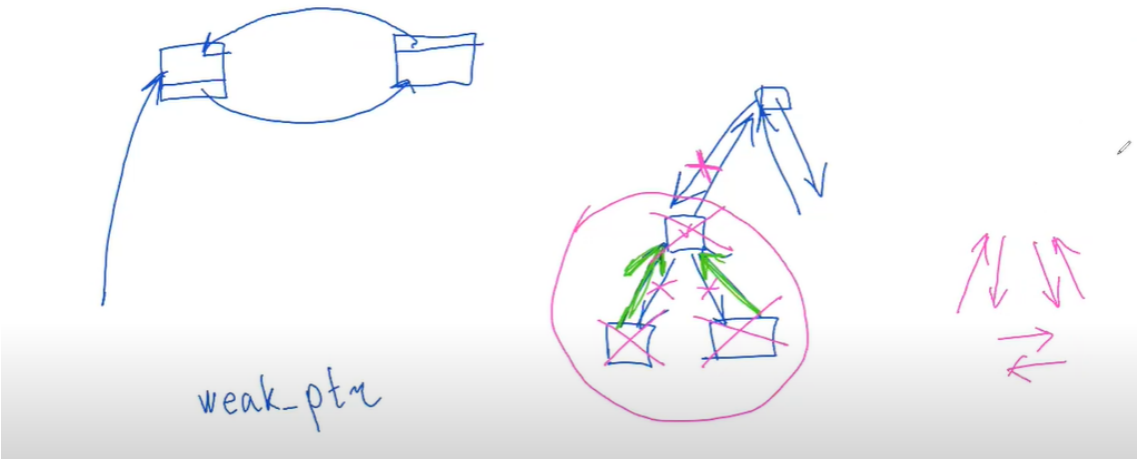
Проблема с гарантиями безопасности в map/unordered_map

Допустим, мы начали перестраивать дерево/хэш таблицу, и у нас выскочило исключение. В такой ситуации контейнер не может вернуть все в первоначальное состояние из-за того, что это весьма затруднительное мероприятие в силу внутреннего устройства этих контейнеров

5.16 Проблема циклических shared_ptr. Класс weak_ptr как решение этой проблемы. Реализация основных методов weak_ptr: конструкторы, деструктор, операторы присваивания, методы expired() и lock(). Модификация реализации shared_ptr для поддержки weak_ptr.

Еще одна проблема - возможная циклическая зависимость (другие языки со сборкой мусора тоже ею страдают).

Допустим мы реализуем двоичное дерево и в какой-то момент хотим удалить поддерево этого дерева. Логично предположить, что все указатели должны удалиться, однако из-за того что внутри поддерева сын указывает на родителя, а родитель на сына, будут оставаться объекты указывающие на вершинку - следовательно, вершинка не удалится, и так со всеми вершинками в удаляемом дереве. Т.о. произошла утечка памяти. Решение проблемы - **weak_ptr**



Это сущность, которая как и shared_ptr хранит указатель на некий объект, однако он им не владеет (просто смотрит) .

Его можно спросить две вещи:

1 Не умер ли еще объект на который ты указываешь

2 Создай новый shared_ptr на объект, на который ты смотришь

Сам weak_ptr разыменовывать нельзя.
Если объект убит, а мы попросили shared_ptr то это будет UB/RE.

Как weak_ptr решает проблему? Можно сделать указатели на родителей слабыми(зеленые стрелки на рисунке)
Тогда если мы обнуляем указатель идущий от родителя к вершине (стрелка, зачеркнутая жирным розовым крестиком), вершина понимает что на нее указывает 0 shared_ptr и 2 weak_ptr. Но weak_ptr-ы не считаются и объект уничтожается. Следом умирают указатели, идущие от вершины к детям (указатели, идущие в противоположную сторону от зеленых), и уничтожаются самые нижние вершины

Правило: если есть двустороняя циклическая зависмость, то хотя бы один указатель (связь) в каждой вершине в ней должен быть weak_ptr-ом. Тогда если хотя бы одна связь из цикла будет разрушена, будет разрушен и весь цикл, причем в правильном порядке

Реализация

```
 template<typename T>
 class weak_ptr{
 private:
         ControlBlock<T>* inner_block = nullptr;
 public:
         weak_ptr(const shared_ptr<T>& p): inner_block(p.inner_block){}; //
constructor

         weak_ptr(const weak_ptr<T>& other):inner_block(other.inner_block){};
//copy
constructor

         weak_ptr& operator = (const weak_ptr<T>& other){
             if (this != std::addressof(other)) {
                 inner_block = other.inner_block;
             }
             return *this;
         }

         weak_ptr(weak_ptr<T>&& other):
         inner_block(std::move(other.inner_block)){};

         weak_ptr& operator = (weak_ptr<T>&& other){
             if (this != std::addressof(other)) {
                 inner_block = std::move(other.inner_block);
             }
             return *this;
         }

         bool expired() const {
             return inner_block->shared_cnt == 0;
         }

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

         }

         ~weak_ptr() {
             if(!inner_block) return;
             --inner_block->weak_cnt;
             if (inner_block->weak_cnt == 0 and inner_block->shared_cnt == 0)
{
                 delete inner_block;
             }
         }

 };
```

shared_ptr with weak_ptr

```
template <typename U>
struct ControlBlock {
    T object;
    size_t count;

    template <typename... Args>
    ControlBlock(size_t count, Args&&... args);
};
```

```
template<typename T>
 class shared_ptr{
 private:
     T* ptr;
     ControlBlock<T>* inner_block = nullptr;

     template<typename  U>
     friend class weak_ptr;

     template <typename U, typename ...Args>
     friend shared_ptr<U> make_shared(Args&& ...args);

     shared_ptr(ControlBlock<T>* cb) : inner_block(cb), ptr(cb->val){};
 public:
     explicit shared_ptr(T* pointer){
         inner_block = new ControlBlock<T>{1, pointer};
         ptr = pointer;
         if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>) {
             ptr->wptr - *this;
         }
     }

     shared_ptr(const shared_ptr<T>& other) {
         ptr = other.ptr;
         ++inner_block->shared_cnt;
     }

     shared_ptr(shared_ptr<T>&& other) {
         inner_block = std::move(other.inner_block);
         other.ptr = nullptr;
     }

     shared_ptr<T>& operator=(const shared_ptr<T>& other) & {
         shared_ptr<T> copy(other);
         std::swap(copy, *this);
         return *this;
     }
     shared_ptr<T>& operator=(shared_ptr<T>&& other) & noexcept {
         inner_block = std::move(other.inner_block);
         other.ptr = nullptr;
         return *this;
     }

     ~shared_ptr() {
         if (!inner_block) return; // no control block
         --inner_block->shared_cnt;
         if (inner_block->shared_cnt == 0) {
             delete ptr; // obj deleted, but not control block
             if (inner_block->weak_cnt == 0 ) {  //if false, then some weak_ptr are looking at obj
                 delete inner_block;
             }
         }
     }

 };
```

5.17 Класс enable_shared_from_this, описание проблемы, которую он решает. Реализация этого класса.

Как получить из тела метода структруры/класса умный указатель на самого себя? Если возникает такая ситуация - т.е мы хотим чтобы класс поддерживал возможность возвращать shared_ptr на себя, то мы не в полях заводим weak_ptr, а обращаемся к некоторой бибилиотечной функции, которая генерит shared_ptr (который начинает делить владение объектом с уже созданными указателями) и возвращает его нам.

CRTP = Curiously Recursive Template Pattern

См. часть реализации в shared_ptr (конструктор)

```
template<typename  T>
class enable_shared_from_this{
private:
    weak_ptr<T> ptr = nullptr;
protected:
    shared_ptr<T> shared_from_this() const {
        return ptr.lock();
    }
};

struct S : public enable_shared_from_this<S> {
    shared_ptr<S> getPointer() const {
        return shared_from_this();
    }
};
```

Определение структры S не требуется для работы enable_shared_from_this<S>.

5.18 Лямбда-функции. Объекты анонимного типа, сгенерированные из лямбда-функций (замыкания), их внутреннее устройство. Правила генерации компилятором полей и методов этих объектов. Особенности копирования и присваивания этих объектов. Особенности захвата полей класса и указателя this в лямбда-функции. Опасность захвата по умолчанию.

Выражение стоящее после [.] называется **замыканием** (closure). Оно является rvalue

Синтаксис

```
std::vector<int> v = {1,6,4,6,3,6};
std::sort(v.begin(), v.end(), [](int x, int y) {
    return std::abs(x - 5) < std::abs(y - 5);
});
```

Можно объект проинициализировать лямбда-функцией, тип объекта - ожидаемо auto.

Тип, который имеет лямбда-функция, генерирует компилятор. Он превращает код в некоторое внутреннее представление и дает имя, которое бы не пересекалось с другими именами в программе.

Давайте попробуем узнать размер объекта f

```
auto f = [](int x, int y) {
    return x < y;
};

//g(f);

std::cout << typeid(f).name() << '\n';

std::cout << sizeof(f) << '\n';
```

Программа выдаст 1 байт. Это происходит потому, что у этого объекта нет никаких полей, а все, что есть - один метод, функция f. Можно считать, что все, что у нас есть - это один пустой функциональный класс, сгенерированный компилятором, в котором определен оператор ()

Что будет, если теперь захватить то-то в лямбда-функцию?

```
int a = 1;

auto f = [a](int x, int y) {
    return x + a < y;
};
```

Теперь размер f увеличился до 4 байт, потому что мы захватили объект. То есть в сгенерированном функциональном классе появилось поле int

Что на самом деле происходит, когда мы что-то захватываем в лямбда-функцию?
Компилятор генерирует под каждую функцию специальный класс, тип которого олицетворяет эту лямбда-функцию, в котором он автоматически определяет оператор круглые скобочки, исходя из параметров, переданных в лямбда-функцию. Все захваченные переменные компилятор делает полями созданного функционального класса, и созданные поля инициализирует тем, то мы захватили

Соответственно, если мы захватим по ссылке

```
int a = 1;

auto f = [&a](int x, int y) {
    return x + a < y;
};
```

то теперь размер f будет 8 байт, потому что теперь полем является ссылка на int, а ссылка на int инициализируется a при создании лямбда-функции

Отсюда понятно, почему поля, которые мы захватили, как копии по значению, неизменны. Это происходит потому что оператор () из лямбда-функций по стандарту константный.

Понятно, почему он так должен делать. Потому что всякая нормальная сортировка имеет право считать компаратор константным. Соотвественно, если компилятор создал поля в этом классе, то он считает их неизменными из-за того, то у нас константный метод.

А вот со ссылками так не работат, потому что const в методе не распространяется на ссылки. Если у нас поле int&, то даже если метод константный, поле все равно может меняться из этого метода, потому что const у метода означает навешивание const справа на тип, а не слева

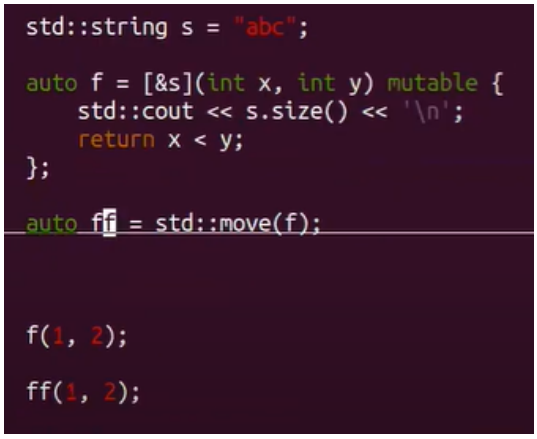
Кроме оператора () компилятор генерирует для функционального класса конструктор копирования (это логично, так как иначе бы нельзя было передавать нашу функцию по значению куда-либо) и мув-конструктор. Первый просто копирует поля, а второе мувает все поля

```
auto f = [a](int x, int y) mutable {
    ++a;
    std::cout << a << '\n';
    return x + a < y;
};

auto ff = f;

f(1, 2);

ff(1, 2);
```

Здесь в обоих случаях выведется 2, изменение a в одной функции не повлияло на a в другой. Если бы a принималось по ссылке, то ответ был бы 2 и 3

```
std::string s = "abc";

auto f = [s](int x, int y) mutable {
    std::cout << s.size() << '\n';
    return x < y;
};

auto ff = std::move(f);

f(1, 2);

ff(1, 2);
```

А здесь мы мувнули строку из f, в результате чего она стала пустой. Выведется 0 и 3

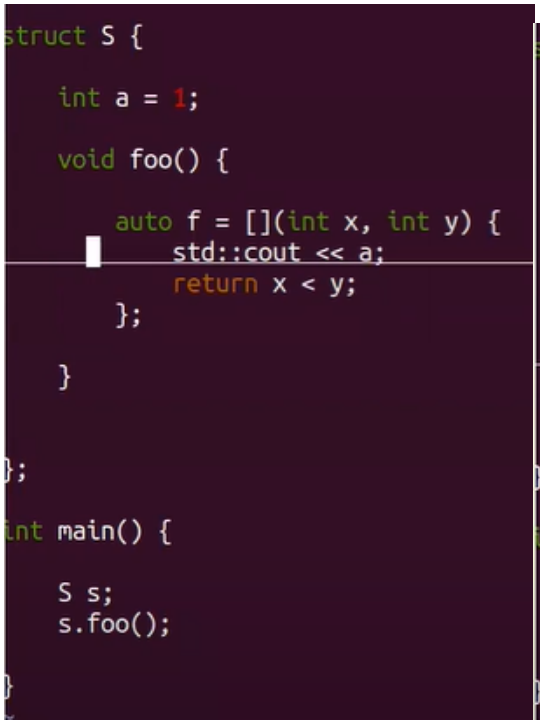


Выведется 3 и 3, поскольку мув ссылки не делает ничего

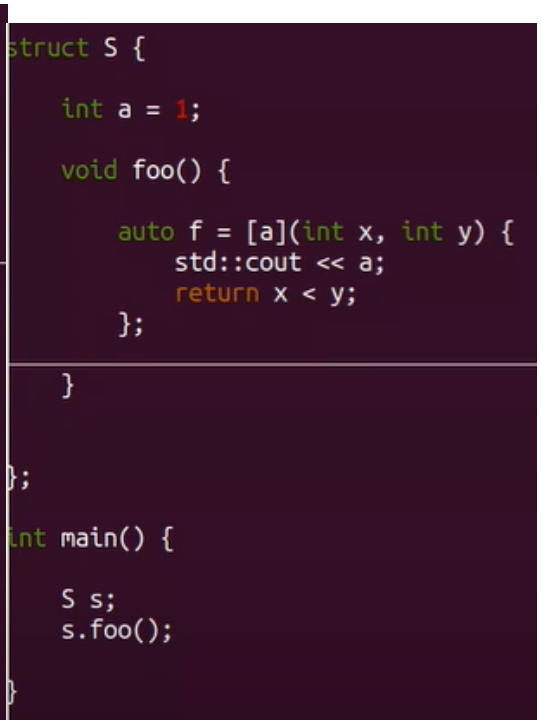
Компилятор не генерирует оператор присваивания, если есть захват! Но начиная с C++20 он может генерировать оператор присваивания, если захвата нет

Начиная с C++20 компилятор генерирует конструктор по умолчанию, до этого выдавал CE

Особенности захвата полей класса и указателя this в лямбда-функции



Не сработает

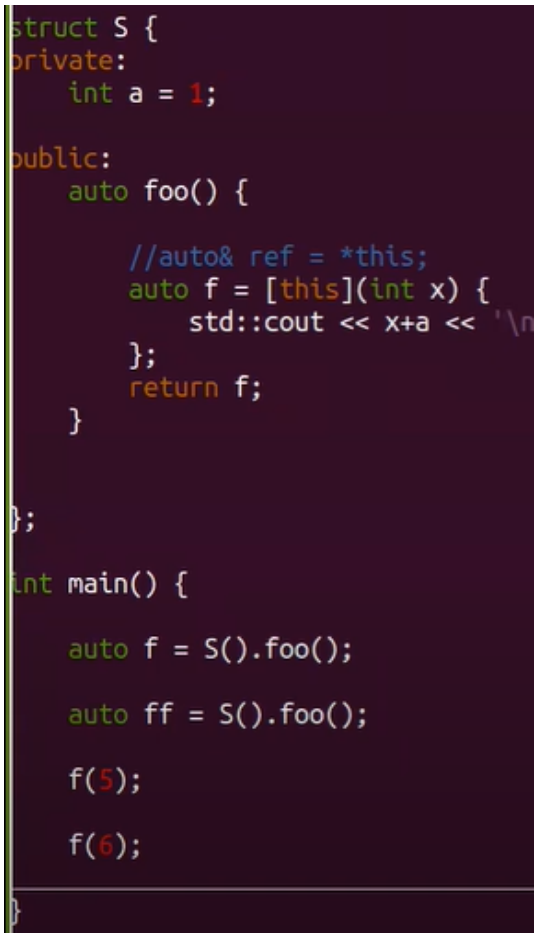


Не сработает

Захватывать можно только локальные переменные, но не поля класса

Но зато можно сделать вот так

В лямбду мы захватываем указатель на текущий объект. Теперь мы можем образаться к полям this из функции



Вот так будет UB, так как функция пережила тот объект, который она захватила

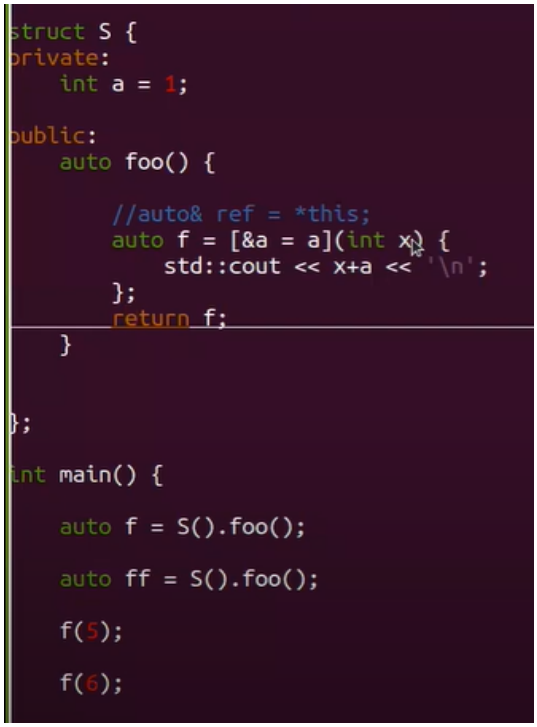
Capture with initialization

Начиная с C++14 появилась возможноть инициализации при захвате

Для компилятора это значит, что мы просим его завести в генерируемом классе поле b, но проинициализировать его посредством a, которое берется из текущей области видимости, включая поля класса

Самое прекрасное, что от нас никто не требует, чтобы поле и то, чем мы его инициализируем, назывались по-разному. То есть такая штука тоже сработает, и это будет не то же самое, что просто захватить a, сейчас не будет никакого UB и все нормально скомпилируется

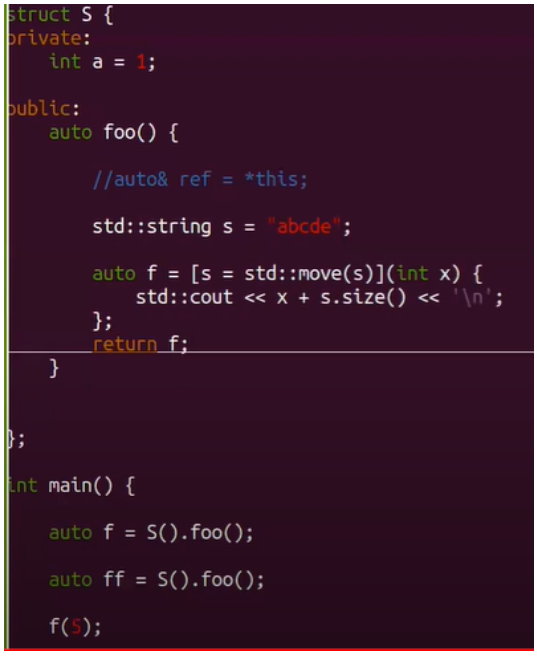
Вот так тоже можно сделать, но мы получим UB, потому что то, чем мы проинициализировали ссылку, умрет раньше, чем функция. И во втором случае также будет UB, по той же причине



```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;
        auto f = [&a = a](int x) {
            std::cout << x+a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);

    f(6);
```

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;
        auto f = [a = &a](int x) {
            std::cout << x + *a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);

    f(6);
```

Еще одна важная вещь, которая появилась - возможность захватывать по rvalue ссылке. Делается это с помощью захвата с инициализацией и std::move(1).
Вот мы умеем захватывать по значению, по ссылке, мувать... а можно по константной сылке? Да!(2)

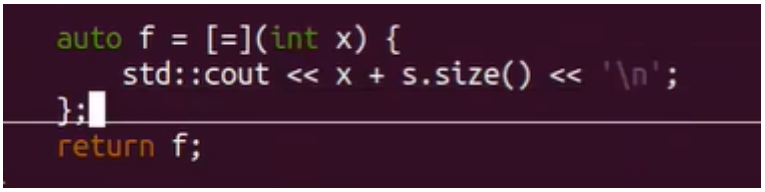


```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [s = std::move(s)](int x) {
            std::cout << x + s.size() << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);
```

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [&s = std::as_const(s)](int x) {
            std::cout << x + s.size() << '\n';
        };
        return f;
    }


};
```

(1) (2)

Есть синтаксис захвата всех переменных сразу. Можно захватить все локальные переменные по значению или по ссылке. **Но так не надо делать!**. Потому что так или иначе мы забываем, что захватили. И тем самым мы получаем доступ к к переменным, которые на самом деле, возможно, уже умерли



```
auto f = [=](int x) {
    std::cout << x + s.size() << '\n';
};
return f;
```

Так не надо

Приведем пример, почему это плохо

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [=](int x) {
            std::cout << x + a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();
```

Вот тут мы как будето захватили все имена по значению, но на деле мы захватили this, то есть захватили еще и a, которое умерло раньше, чем наша функция, и в итоге получилось UB
Еще хуже захват по ссылке. Будет такое же UB

```
auto f = [&](int x) {
    std::cout << x + a << '\n';
};
```

Так не надо

Что можно сделать еще... Сказать "захвати все по ссылке, кроме s"

```
auto f = [&, s](int x) {
    std::cout << x + a << '\n';
};
```

"захвати все по значению, кроме s"

```
auto f = [=, &](int x) {
    std::cout << x + a << '\n';
};
```

5.19 Класс std::function, его основные методы и примеры использования. Опишите идею внутреннего устройства std::function: каким образом достигается возможность по ходу работы подменять хранящийся в ней функциональный объект? (Принимается любая работающая идея.)

std::function - это тип, который позволяет нам инициализировать себя любым объектом, который являтся callable.

Представьте, что у вас есть какой-то класс или какая-то функция, у которой одним из параметров должна быть другая функция. Например, мы пишем сортировку, и одним из ее параметров является компаратор. Но мы хотим, чтобы в качестве копаратора можно было передать не что угодно, а только вещи типа, которые вызываемы, то есь они callable, в терминах питона. Такие, что их можно вызыать от таких-то типов с такими-то параметрами

Ну, или например, мы пишем A*. И у нас там в качестве параметра используется какая-то функция (расстояния, эвристики). И эта функция должна быть полем класса. Какого типа должно быть это поле? Делать кучу шаблонных параметров на каждую функцию в классе очень странно, это некрасиво.

В c++11 появился такой тип, который называется std::function, который как раз позволяет нам хранить любой объект, имеющий оператор () именно от тех типов, которые мы ему сказали. Например, мы хотим, чтобы у нас была функция, принимающая два инта, возвращающая буль. Тогда пишем следующим образом:

```
std::function<bool(int, int)> f;
```

Этот объект можно проинициализировать любым объектом, имеющим оператор круглые скобки от двух int, возвращающий bool. В том числе и лямбда-функцией, в том числе указателем на сишную функцию

```
std::function<bool(int, int)> f;

f(1, 2);
```

Если мы пытаемся вызвать функцию, которая ничем не проницализировна, получаем исключение bad_function_call

```
f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
}
```

Пример инициализации f

Здесь работает приведение типов. Если мы напишем так, то все сработает:

```
std::function<int(int, int)> f;

//f(1, 2);

f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
};

f(1, 2);
```

```
std::function<int(bool, int)> f;

//f(1, 2);

f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
};

f(1, 2);
```

```
struct S {

    bool operator()(int x, int y) const {
        std::cout << "Hello!\n";
        return x > y;
    }

};


int main() {

    std::function<bool(int, int)> f;

    //f(1, 2);

    f = [](int x, int y) {
        std::cout << "Hi!\n";
        return x < y;
    };

    f(1, 2);

    f = S();

}
```

Переприсваивание работает. Первоначальный объект уничтожится, а новый положится

```
bool g(int x, int y) {
    std::cout << "Blablabla!\n";
    return x == y;
}


int main() {

    std::function<bool(int, int)> f;

    //f(1, 2);

    f = [](int x, int y) {
        std::cout << "Hi!\n";
        return x < y;
    };

    f(1, 2);

    f = S();

    f(3, 4);

    f = &g;

```

Адрес сишной функции тоже скормится без проблем. Если убрать знак взятия адреса, то все тоже корректно будет работать

Внутренне устройство

Member functions

(constructor)	constructs a new std::function instance (public member function)
(destructor)	destroys a std::function instance (public member function)
operator=	assigns a new target (public member function)
swap	swaps the contents (public member function)
assign (removed in C++17)	assigns a new target (public member function)
operator bool	checks if a target is contained (public member function)
operator()	invokes the target (public member function)

Что внутри себя хранит function. Внутри нее есть некоторый класс Manager с шаблонным параметром F (тип функтора), в котором определены статические методы на каждое возможное действие с функтором (сделать копию, сделать инициализацию и т.д.).
Типы функторов:
• Указатель на C-style функцию;
• Обычный функциональный объект (т.е. объект, обладающий оператором () с соответствующими типами аргументов);
• Замыкание (то есть объект, созданный лямбда-выражением);

В Function хранятся указатели на эти методы (точнее, указатель на один метод с дополнительным параметром OperationType, с помощью которого можно выбирать нужный метод).

Когда мы инициализируем функцию чем-то новым, она вызывает у своего старого диспетчера (manager) метод "уничтожить", при этом, возможно, делается static_cast к нужному F, а потом создается указатель на нового диспетчера, и у него вызывается метод initialize (диспетчер запоминается)

Немного про идею реализации function

Класс Function должен обладать следующей функциональностью:
• Конструктор по умолчанию;
• Конструктор от Callable-объекта. Объект должно быть можно отдать в этот конструктор как в виде lvalue (и тогда Function должна скопировать его содержимое в себя), так и в виде rvalue (тогда Function должна мувнуть в себя его содержимое). Если принятый объект не является Callable с нужными аргументами, попытка создать Function от него должна приводить к CE.
• Конструктор копирования, конструктор перемещения, операторы присваивания (copy и move), деструктор.
• Оператор () с соответствующими аргументами, позволяющий вызвать хранимый в Function объект как функцию. Если там сейчас не хранится никакого объекта, нужно бросить исключение.

Кроме того:
• Function должна быть легковесным объектом. А именно, sizeof(Function) должен не превосходить 32 байт (ибо sizeof(std::function) в контесте именно такой).
• Предыдущий пункт означает, что если Callable-объект, который нужно сохранить в Function, достаточно большой, то под него надо выделять динамическую память. Это можно делать напрямую с помощью new/delete, аллокатором в этой задаче пользоваться необязательно.
• Обращения к new/delete надо по возможности экономить. Если новый Callable-объект можно положить на то же место, где лежал старый, то не надо делать перевыделение памяти.

• Для продвинутого потока: Если Callable-объект является обычным указателем на функцию, или указателем на метод, или чем-либо другим, по размеру не превосходящим max(void(*)(), void(C::*)()) (на практике это 16 байт), то динамическая память под него выделяться не должна! Такие объекты Function должна уметь хранить внутри своих полей, т.е. на стеке.
• Для основного потока: Если Callable-объект достаточно мал по размеру (не превосходит 16 байт), то динамическая память под него выделяться не должна! Такие объекты Function должна уметь хранить внутри своих полей, т.е. на стеке.

5.20. Реализуйте проверку числа N на простоту в compile-time с асимптотикой $O(\sqrt{N})$ с помощью шаблонной рекурсии (без использования constexpr, а также математических функций стандартной библиотеки).

Все что нам нужно - это реализовать метафункцию для поиска квадратного корня и подставить его в `IsPrimeHelper` из билета 4.11. вместо N-1 (так как нам достаточно проверить только делимость на числа $1 \dots \sqrt{N}$). Корень находим бинарным поиском за логарифм, значит итоговая асимптотика $O(\log N + \sqrt{N}) = O(\sqrt{N})$

```
template<size_t N, size_t LO=1, size_t HI=N>
struct Sqrt { // используем бинарный поиск
private:
  // вычисляем середину округленную вверх
  static const size_t mid = (LO + HI + 1) / 2;
public:
  // сравниваем N с квадратом середины интервала
  static const size_t value = (N < mid * mid) ? Sqrt<N, LO, mid - 1>::value
                                              : Sqrt<N, mid, HI>::value;
};

// частичная спеииализация для случая LO = HI (нашли ответ)
template<size_t N, size_t M>
struct Sqrt<N, M, M> {
    static const size_t value = M;
};

template <size_t N>
struct IsPrime { // просто подставляем корень из N в IsPrimeHelper
    static const bool value = IsPrimeHelper<N, Sqrt<N>::value>::value;
};

template <>
struct IsPrime<1> { // отдельно случай для 1
    static const bool value = false;
};
```

5.21. Реализуйте метафункцию has_method, позволяющую проверить, присутствует ли у класса T метод с заранее заданным названием от данных типов аргументов. Объясните, как работает ваша реализация. Для чего в ней нужна функция declval и что эта функция из себя представляет? Для чего и в каких местах STL используется метафункция has_method?

Рассмотрим функцию has_method на примере метода construct

```
// T - класс в котором проверяем метод, Args - аргументы метода
template <typename T, typename... Args>
struct has_method_construct {
private:
    template<typename TT, typename... AArgs>
    static auto f(int) -> decltype(declval<TT>().construct(declval<AArgs>()
    ...), int());
    // возвращаемый тип - int, так как оператор запятая возвращает последний операнд

    template<typename...>
    static char f(...);
public:
    static const bool value = std::is_same_v< decltype(f<T, Args...>(0)),
    int >;
    // будет истина только когда выбралась первая версия функции
};
```

Объяснение: Как и все подобные функции, has_method работает благодаря SFINAE. Когда мы вызываемся от 0, компилятор пытается выбрать первое объявление f как более частное (возвращаемый тип int), но если искомого метода у класса нет, то сработает SFINAE и он перейдет ко второму варианту (возвращаемый тип char).

Зачем делать функции f шаблонными? Если не сделать и просто воспользоваться T и Args, то шаблонные параметры зафиксируются в момент инстанцирования класса и SFINAE не сработает.

Функция declval: Для того чтобы все заработало нам понадобилось получить выражения типа T и Args. Возникла проблема: не у всех классов есть конструкторы по умолчанию. Чтобы избавится от этой проблемы мы воспользовались функцией declval. Что же она делает?

```
template<typename T>
T&& declval() noexcept;
```

Двойной амперсанд нужен, чтобы функция работала для incomplete types (тех у которых нет определения. Объекты такого типа нельзя создать, а вот ссылку на них - можно) + тип value не испортится (если T было lvalue/rvalue оно таким и останется).

Заметим, что у функции нет тела, нам оно и не нужно, так как она используется только для проверок в compile-time. Таким образом, можно сказать, что declval - это противоположность decltype.

$$\text{type T} \underset{\text{declval}}{\overset{\text{decltype}}{\longleftrightarrow}} \text{expression of type T}$$

Применение: В STL функция has_method используется, например, в allocator_traits, когда пытаемся определить, есть ли пользовательский метод construct/destroy/etc. или нужно взять дефолтный.

5.22. Реализуйте метафункции is_constructible, is_copy_constru is_move_constructible. Объясните, как работают ваши реализации. Для чего в них нужна функция declval и что она из себя представляет?

```cpp
// T - класс в котором проверяем метод, Args - аргументы метода
template <typename T, typename... Args>
struct is_constructible {
private:
    template<typename TT, typename... AArgs>
    static auto f(int) -> decltype(TT(declval<AArgs>()...), int());
    // возвращаемый тип - int, так как оператор запятая возвращает последний операнд

    template<typename...>
    static char f(...);
public:
    static const bool value = std::is_same_v< decltype(f<T, Args...>(0)),
    int >;
    // будет истина только когда выбралась первая версия функции
};

template<typename T>
using is_copy_constructible = is_constructible<T, const T&>

template<typename T>
using is_move_constructible = is_constructible<T, T&&>
```

Объяснение: Реализация практически ничем не отличается от has_method, объяснение аналогичное. В первой версии пытаемся вызвать конструктор от нужных аргументов, если его нет, то сработает SFINAE. (Про declval см. билет 5.21.)

По определению copy и move конструкторы - это конструкторы от const T& и T&& соответственно, поэтому просто выражаем метафункции для них через is_constructible.

5.23. Реализуйте метафункцию is_nothrow_move_constructible и с помощью нее реализуйте функцию move_if_noexcept. Объясните, как работает ваша реализация. Почему нельзя наивно выразить is_nothrow_move_constructible как is_move_constructible_v<...> && noexcept(...)?

Нам понадобится вспомогательный класс `integral_constant`, отвечающий за compile-time константы

```
template<typename T, T v>
struct integral_constant {
    static const T value = v;
};
```

Через него можно выразить две важных константы:

- `true_type = integral_constant<bool, true>`
- `false_type = integral_constant<bool, false>`

```
template<typename T>
struct is_nothrow_move_constructible {
private:
    template<typename TT>
    static auto f(int) -> integral_constant<bool,
                                            noexcept(TT(declval<TT>()))>;
    // более понятный вариант:
    // -> std::conditional_t<noexcept(TT(declval<TT>()), true_type, false_type>

    template<typename...>
    static auto f(...) -> false_type;
public:
    static const bool value = decltype(f<T>(0))::value;
};

template<typename T>
const bool is_nothrow_move_constructible_v =
            is_nothrow_move_constructible<T>::value;

template<typename T>
auto move_if_noexcept(T& x) -> std::conditional_t<
        is_nothrow_move_constructible_v<T>, T&&, const T&> {
    return std::move(x);
}
```

Объяснение:

1. У нас нет move конструктора. Тогда по SFINAE выберется вторая версия f и мы получим false
2. Есть move конструктор, но он не noexcept. Тогда возвращаемый тип первой версии f станет `false_type` и мы получим ответ false
3. Move конструктор есть и он noexcept. Аналогично пункту 2 получим ответ true.

Принцип работы move_if_noexcept очевиден (муваем если есть noexcept move конструктор, иначе - возвращаем константную ссылку)

Неправильная реализация: Почему нельзя просто сделать так?

```
template <typename T>
auto move_if_noexcept(T& x) -> std::conditional_t<
        is_move_constructible_v<T> && noexcept(T(declval<T>())),
        T&&, const T&> {
    return std::move(x);
}
```

Мы привыкли к тому, что вторая часть конъюнкции не вычисляется, но забыли о том, что она всегда компилируется. Если у нас не будет move конструктора, то часть с `noexcept(...)` попросту не скомпилируется, и даже `is_move_constructible_v<T>` нас не спасет.

5.24. Реализуйте метафункцию is_base_of<T, U>, позволяющую проверить, является ли класс U наследником класса T (в том числе приватным). Объясните, как работает ваша реализация.

```
namespace details {
    template <typename B>
    auto f(B*) -> true_type;

    template <typename...>
    auto f(...) -> false_type;

    template <typename B, typename D>
    auto test(int) -> decltype(f<B>(declval<D*>()));

    template <typename...> // сюда попадем только если наследование было приватным
    auto test(...) -> true_type;
}

template<typename B, typename D>
struct is_base_of: integral_constant <bool,
        std::is_class_v<B> && std::is_class_v<D> && // проверяем что это классы
        // вариант не работающий с приватным наследованием:
        // decltype(details::f<B>(std::declval<D*>()))::value> {};
        decltype(details::test<B, D>(0))::value> {};
```

Объяснение: Мы пытаемся подставить предполагаемого наследника вместо родителя. Если у нас не получается, то должно сработать SFINAE и перекинуть нас в функцию с возвращаемым значением `false_type`. Если у нас наследование приватное, то одних f нам недостаточно (проверка приватности произойдет после выбора версии для перегрузки, то есть мы уже выберем первую версию, проверим приватность и получим CE).

Добавляем еще один уровень SFINAE с test. В случае, когда наследования нет/оно не приватное test будет действовать так же, как наш первый вариант с f. Если наследование приватное, f не сможет скомпилироваться, а значит по SFINAE нам приется перейти во вторую версию перегрузки test, которая имеет возвращаемый тип `true_type`

Вопрос 25

Реализация с cppreference (ну почти)

```
#include <type_traits>

namespace detail {

template <typename T>
auto detect_is_polymorphic(int) ->
     decltype(dynamic_cast<const void*>(std::declval<T*>()), std::true_type());

template <typename...>
std::false_type detect_is_polymorphic(...);

}

template <class T>
struct is_polymorphic :
    std::bool_constant<std::is_class_v<T> &&
        decltype(detail::detect_is_polymorphic<T>(0))::value> {};
```

Реализация работает так: класс T является полиморфным, если T* можно динамически привести к const void*. Эта идея и используется в реализации. Если dynamic_cast возможен, то будет выбрана первая версия (и возвращаемый тип - std::true_type). Если нет, то будет Substitution Failure, и благодаря SFINAE будет выбрана вторая версия с возвращаемым типом std::false_type.

5.26. Реализуйте структуру make_index_sequence<N>, которая бы представляла из себя index_sequence <%последовательность от 0 до N-1%>.

`index_sequence` - в некотором смысле compile-time массив - структура, шаблонными параметрами которой являются size_t, в каком-то смысле она их «хранит» (`integer_sequence` хранит int). Реализуем `make_index_sequence`.

```
template <size_t... Ints>
struct index_sequence {};

template <typename T, size_t N>
struct push_back; // сама структура нам не нужна, нужна только специализация

template <size_t N, size_t... Ints>
struct push_back<index_sequence<Ints...>, N> {
    using type = index_sequence<Ints..., N>;
}; // добавление числа N в конец index_sequence из Ints

template <size_t N>
struct make_index_sequence_s {
    using type = typename push_back<
            typename make_index_sequence_s<N-1>::type,
            N-1>::type; // добавляем N-1 к предыдущему результату
};

template <>
struct make_index_sequence_s<0> {
    using type = index_sequence<>; // база - пустая последовательность
};

template <size_t N>
using make_index_sequence = typename make_index_sequence_s<N>::type;
```

Для проверки того, что все работает, можно запустить следующую строчку:
`static_assert(std::is_same_v< make_index_sequence<3>, index_sequence<0, 1, 2> >);`
Замечание: Пример использования этих структур приведен в билете 5.27.

5.27. Объясните понятие рефлексии в программировании. Реализуйте функцию detect_fields_count<S>, позволяющую для данной структуры S, допускающей агрегатную инициализацию, узнать количество полей в этой структуре.

Рефлексия в программировании: Процесс, во время которого программа начинает задаваться вопросами о своем коде («а правда ли, что класс в котором я нахожусь называется так?», «а правда ли, что этот метод приватный?» и т. д.).

Реализуем функцию `detect_fields_count<S>` позволяющую узнать количество полей структуры S (с условиями из заголовка билета)

```
struct ubiq { // структура, которая приводится к любому типу
    template<typename T>
    operator T();
};

template <int N> // зачем шаблонный параметр? Ответ после кода :)
using ubiq_constructor = ubiq;

template <class T, int I0, int... I> // более частная, 1 аргумент "откушен"
auto detect_fields_count(index_sequence<I0, I...>)
 -> decltype(T{ubiq_constructor<I0>{}, ubiq_constructor<I>{}... }, int() ) {
    return sizeof...(I) + 1;
}

template <class T, int... I>
int detect_fields_count(index_sequence<I...>) {
    return detect_fields_count<T>(make_index_sequence<sizeof...(I) - 1>{});
}

int main() {
    std::cout << detect_fields_count<S>(make_index_sequence<100>{});
}
```

Объяснение работы: При таком запуске выберется первая версия как более частная (так как первый аргумент от хвоста откушен). Мы пытаемся сконструировать наш тип от такого же количества аргументов, сколько size_t нам передалось в шаблон (а вот и ответ на вопрос в коде: фиктивный шаблонный параметр нужен, чтобы мы смогли сделать развертку по всем элементам из хвоста). Если у нас не получилось, то мы взяли взяли слишком много аргументов. По SFINAE мы перепрыгнем во вторую версию функции, в которой мы просто вызываемся от `index_sequence`, длина которой на 1 меньше текущей.

Таким образом, если у нашей структуры ≤ 100 полей рано или поздно будет выбрана первая функция, которая выведет их точное количество.

Замечание: Мещерин на лекции вызывался от 100. Имеет смысл вызваться от sizeof(S), чтобы точно превзойти число полей (проблема этого варианта: можем уйти в очень глубокую рекурсию).

extra1. Указатели на функции, синтаксис их объявления и примеры использования. Указатели на члены, синтаксис объявления и использования. Операторы “точка со звездой” и “стрелочка со звездой”. Указатели на методы, синтаксис использования. Практический пример: своп начала и конца отрезка в зависимости от параметра функции.

```
T f(int, int){}

int main(){
    T (*pf)(int, int)= &f;
}
```

Объявление указателя на функцию. Типо pf будет T(*)(int, int) Указателям можно присваивать указатели других функций.

Присваивание функции указателю на функцию

```
int boo()
{
  return 7;
}

int doo()
{
  return 8;
}

int main()
{
  int (*fcnPtr)() = boo; // fcnPtr указывает на функцию boo()
  fcnPtr = doo; // fcnPtr теперь указывает на функцию doo()

  return 0;
}
```

Вызов функции через указатель на функцию

Способ 1. Явное разыменование.

```
int boo(int a)
{
    return a;
}

int main()
{
    int (*fcnPtr)(int) = boo; // присваиваем функцию boo() указателю fcnPtr
    (*fcnPtr)(7); // вызываем функцию boo(7), используя fcnPtr

    return 0;
}
```

Способ 2. Неявное разыменование.2

```
int boo(int a)
{
    return a;
```

```
}

int main()
{
    int (*fcnPtr)(int) = boo; // присваиваем функцию boo() указателю fcnPtr
    fcnPtr(7); // вызываем функцию boo(7), используя fcnPtr

    return 0;
}
```

Указатели на функции-члены

.* и ->*

Указатели на методы классов

extra2. Размещение объекта в памяти в случае ромбовидного наследования от полиморфных типов. Независимость проблемы ромбовидного наследования от наличия виртуальных функций. Количество указателей на vtable. Устройство таблицы для Mother-in-Son и для Father-in-Son, разница между этими таблицами. Понятие top_offset. Устройство vtable в случае виртуального наследования с виртуальными функциями. Значения virtual_offset и top_offset.

Классы, у которых есть виртуальные функции, называются полиморфными. Пусть есть типичное ромбовидное наследование, но только все классы имеют virtual функцию.

```
struct Granny {
    virtual void fg();
    int g;
};

struct Mother : public Granny {
    virtual void fm();
    int m;
};
struct Father : public Granny {
    virtual void ff();
    int f;
};
struct Son : Mother, Father {
    virtual void fs();
    int s;
};
```

Заметим, что если в такой конструкции у Son попытаться вызвать fg(), то проблема ромбовидного наследования останется, так как компилятор:

Сначала выбирает всех кандидатов на fg()

Потом понимает, как из них — наилучшая перегрузка

Затем проверяет приватность

И только потом заканчивает компиляцию и исполняет программу

Не трудно заметить, что, независимо от виртуальности функции gf(), CE будет уже на втором этапе, так как есть две идентичные перегрузки — из мамы и из папы.

Как мы помним без виртуальности их расположение в памяти было бы:

$[g][m][g][f][s]$

Но теперь у нас должны появиться указатели на vtable, вероятнее всего, где-нибудь в начале. Но сколько же конкретно их нам понадобится? 1? 2? 3? Попробуем реализовать лишь с одним:

$[vptr][g][m][g][f][s]$

Но что же тогда будет, если мы кастанем сына к отцу (Son sptr; Father* fptr = sptr;)? По тем правилам, что мы изучали ранее мы должны сузиться до [g][f][s], у которого vptr в начале нету. Но если мы захотим вызвать ff() — компилятор попробует найти vptr для виртуальной функции, но не сможет. Потому корректная реализация такая:

$[vptr1][g][m][vptr2][g][f][s]$

Но остается загадка: будут ли vptr1 и vptr2 указывать на одну и ту же таблицу? Ответ — нет. Но почему? Пусть vptr1 и vptr2 равны, но давайте вспомним о dynamic_cast. Если указатели одинаковы, то type_id будет один и тот же — нет никакого способа различить Mother и Father. Как следствие dynamic_cast не сможет понять, в чем разница между кастом к Mother и кастом к Father.

Разберемся чуть подробнее с устройством таблицы для Mother-in-Son и для Father-in-Son. Вспомним устройство vtable:

$[type\_info][\&S :: fg]...$

Где vptr указывает ровно на первый блок [&S::fg]. Собственно разница между Mother-in-Son и Father-in-Son будет в том, что в одном классе написано "Я — Mother, чтоб кастануть меня к Mother — делать ничего не надо, а чтоб кастануть к Father — сдвинься на 16 байт вправо". И аналогично во втором.

Теперь рассмотрим случай виртуального наследования:

```
struct Granny {
    virtual void foo();
    int g;
};

struct Mother : virtual public Granny {
    int m;
};
struct Father : public virtual Granny {
    int f;
};
struct Son : Mother, Father {
    int s;
};
```

Тогда устройство в памяти такое:

$[vptr][m][vptr][f][s][g]$

Как же теперь устроен vtable? Вот так:

$[virtual\_offset][top\_offset][type\_info][foo]$

Что же такое top_offset? Это то, насколько байт вправо сдвинута текущая функция от начала. То есть для Father это было бы 16, а для Mother — 0. То есть это помогает понять, куда двигаться, если нас кастуют к другому классу в иерархии, или хотят вызвать функцию у Granny. Для virtual_offset — идея аналогична. Это число, которое надо пройти, чтоб дойти до виртуальных классов.

P.S.: на лекции было сказано, что это устройство для g++. Для других компиляторов, вероятнее всего, если и не тоже и самое, то что-то аналогичное.

extra3 Проблема вызова виртуальных функций в конструкторах и деструкторах. Пример ошибки pure virtual function call. Проблема с аргументами по умолчанию в виртуальных функциях.

Вы не должны вызывать виртуальные функции во время работы конструкторов или деструкторов, потому что эти вызовы будут делать не то, что вы думаете, и результатами их работы вы будете недовольны.

Предположим, что имеется иерархия классов для моделирования биржевых транзакций, то есть поручений на покупку, на продажу и т. д. Важно, чтобы эти транзакции было легко проверить, поэтому каждый раз, когда создается новый объект транзакции, в протокол аудита должна вноситься соответствующая запись. Следующий подход к решению данной проблемы выглядит разумным:

```
1
2  class Transaction { // базовый класс для всех
3      public: // транзакций
4          Transaction();
5          virtual void logTransaction() const = 0; // выполняет зависящую от типа
6          // запись в протокол
7          ...
8  };
9
10 Transaction::Transaction() // реализация конструктора
11 { // базового класса
12     ...
13     logTransaction();
14 }
15
16 class BuyTransaction: public Transaction { // производный класс
17     public:
18         virtual void logTransaction() const = 0; // как протоколировать
19         // транзакции данного типа
20         ...
21 };
22
23 class SellTransaction: public Transaction { // производный класс
24 public:
25     virtual void logTransaction() const = 0; // как протоколировать
26     // транзакции данного типа
27     ...
28 };
```

Посмотрим, что произойдет при исполнении следующего кода: `BuyTransaction b;`

Ясно, что будет вызван конструктор BuyTransaction, но сначала должен быть вызван конструктор Transaction, потому что части объекта, принадлежащие базовому классу, конструируются прежде, чем части, принадлежащие производному классу. В последней строке конструктора Transaction вызывается виртуальная функция logTransaction, тут-то и начинаются сюрпризы. Здесь вызывается та версия logTransaction, которая определена в классе Transaction, а не в BuyTransaction, несмотря на то что тип создаваемого объекта – BuyTransaction. Во время конструирования базового класса не вызываются виртуальные функции, определенные в производном классе. Объект ведет себя так, как будто он принадлежит базовому типу. Короче говоря, во время конструирования базового класса виртуальных функций не существует.

Есть веская причина для столь, казалось бы, неожиданного поведения. Поскольку

конструкторы базовых классов вызываются раньше, чем конструкторы производных, то данные-члены производного класса еще не инициализированы во время работы конструктора базового класса. Это может стать причиной неопределенного поведения и близкого знакомства с отладчиком. Обращение к тем частям объекта, которые еще не были инициализированы, опасно, поэтому C++ не дает такой возможности.

Есть даже более фундаментальные причины. Пока над созданием объекта производного класса трудится конструктор базового класса, типом объекта является базовый класс. Не только виртуальные функции считают его таковым, но и все прочие механизмы языка, использующие информацию о типе во время исполнения (например, описанный в правиле 27 оператор dynamic_cast и оператор typeid). В нашем примере, пока работает конструктор Transaction, инициализируя базовую часть объекта BuyTransaction, этот объект относится к типу Transaction. Именно так его воспринимают все части C++, и в этом есть смысл: части объекта, относящиеся к BuyTransaction, еще не инициализированы, поэтому безопаснее считать, что их не существует вовсе. Объект не является объектом производного класса до тех пор, пока не начнется исполнение конструктора последнего.

То же относится и к деструкторам. Как только начинает исполнение деструктор производного класса, предполагается, что данные-члены, принадлежащие этому классу, не определены, поэтому C++ считает, что их больше не существует. При входе в деструктор базового класса наш объект становится объектом базового класса, и все части C++ – виртуальные функции, оператор dynamic_cast и т. п. – воспринимают его именно так.

Пример кода, вызывающего pure virtual function call:

```
#include <iostream>

class Base
{
 public:
  Base() { init(); }
  ~Base() {}

  virtual void log() = 0;

 private:
  void init() { log(); }
};

class Derived: public Base
{
 public:
  Derived() {}
  ~Derived() {}

  virtual void log() { std::cout << "Derived created" << std::endl; }
};

int main(int argc, char* argv[])
{
  Derived d;
  return 0;
}
```

Фулл по ссылке

беды с башкой (параметрами по умолчанию)

TL;DR

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void Foo (int n = 10) {
        cout << "A::Foo, n = " << n << endl;
    }
};

class B : public A {
public:
    virtual void Foo (int n = 20) {
        cout << "B::Foo, n = " << n << endl;
    }
};

int main() {
    A * pa = new B ();
    pa->Foo ();

    return 0;
}
```

выдаст `B::Foo, n = 10`, потому что компилятор строго следует стандарту языка, предписывающему подставить в код вызова функции значения параметров по умолчанию исходя из статического типа указателя, по которому осуществляется вызов виртуальной функции. Это A * в нашем случае, а значит значения параметра будет взято из декларации функции A::Foo.

extra4. Выражения свертки (fold expressions), их синтаксис и принцип работы. Реализация функции print с помощью fold-expressions. Реализация структуры is_homogeneous с помощью fold expressions. Автоматический вывод типа для шаблонов классов в C++17, пример с вектором. Явное задание правил вывода (deduction guides) для шаблонов классов в C++17, примеры.

Fold expressions

Для всех элементов пакета можно сделать операцию

```
template <typename T>
void print(const Args&... args) {
  (std::cout << ... << args) << '\n';
}
```

Можно писать следующее `( ... < args );` где вместо $<$ любой бинарный оператор. Это Fold expression, у него есть 4 типа:

`( ... < args );` левоассоциативно

`( args < ... );` правоассоциативно

`( x < ... < args );` лево и еще аргумент в начале

`( args < ... < x );` право и аргумент в конце

```
template <typename Hed, typename... Tail>
struct is_homogeneous {
  static const bool value  (std::is_same_v<Head, Tail> && ...);
};
```

Компилятор проверяет, что Head равен каждому элементу Tail

Deduction guides

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

}

int main(){
  int x = 0;
  int& y = x;
  f(y);
}
```

Так как я пишу `f(T x)`, то компилятор захочет принимать по значению

Как кстати проверить что `T` это `int` а не `int&`? Можно сделать `T z = 5;` или `sizeof(T)` или `std::is_reference_v` (в библиотеке `type_traits`)

```
template <typename T>
class C {
  C() = delete;
};

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

А можно так, тут будет конечно CE потому что в compile-time можно узнать про T, потому что вызвется ошибка. В обычной ситуации `const` тоже отброситься, но если сделать `void f(T& x)` то `f(const int&)` сохранит `const`. Если надо вызвать от конкретного типа то так можно и писать `f<int&>(x)`

```
#include <functionals>

int x = 0;
f(std::ref(x));
```

Это такая обертка, считается что это типо ссылка, `reference_wrapper` (но ему нельзя присваивать старые значения(для этого нужно написать `x.get()`), а можно другой `reference_wrapper`). Но у него нет конструктора по умолчанию и нет операций как у указателя.. Но при этом можно сделать ветокр от него (в отличие от ссылки)

Можно с вектором еще сделать так: `std::vector v{1, 2, 3, 4, 5};`

```
template<typename T>
    struct S {
        S(T x) {
            C<T>();
        }
  };

S(const char*) -> S<std::string>;
```

Это user-defined deduction rule, если вызвать `S("abc")` то он будет воспринимать это как строку. При этом deduction guides могут быть и шаблонными

extra5. Выравнивания. Оператор alignof и спецификатор alignas, их действие и пример использования. Основные правила выравнивания стандартных типов. Особенности выравнивания при выделении памяти стандартной функцией operator new. Функция std::aligned_alloc, ее предназначение.

Alignment - минимальная степень двойки, такая что только с адресов кратных этой степени двойки можно класть объект этого типа - выравнивание.

Выравнивание структуруы - выравнивание наибольшого поля

```
struct S {
    int x;
    double y;
    int z;
}

int main() {
    std::cout << sizeof(S);
}
```

Будет 24 байта: 4 на инт, 4 на выравнивание дабла, 8 на дабл, 4 на инт и 4 на выравнивание структуры (чтобы заканчивалась делящимся на 8).

`std::cout << alignof(S);` - как раз минимальная степень двойки, чтобы положить. Здесь 8 (потому что дабл).

Можно написать:

```
struct alignas(8) S {
    int x;
    int y;
};
```

Проблема может быть такая: когда в аллокаторе вызывается `new(count * sizeof(T))` может быть не учтено выравнивание. Правда malloc умная функция и выделит с учетом выравнивания для стнадартных типо (обычно это 8 байт, но мб и больше) - `std::max_align_t` .

Но проблема может быть если мы собственноручно сделали `alignas` больше чем максимальный в стандарте, и тогда будет UB.

Есть `aligned_alloc` - в который выравнивание: `aligned_alloc(alignof(S), n)` - в операторе new.

Это бывает нужно если есть мощный комп и за одну процессорную инструкцию можно сделать действие - за раз 32 байтное число, или сделать операцию сразу с 4 8-байтными числами. Или наоборот для char можно ходить выравнивание 1.

extra6. Класс std::variant и идея его реализации. Каким образом происходит выбор подходящего конструктора при создании variant? Каким образом при уничтожении variant вызывается деструктор нужного типа?

pair, tuple, variant, any, optional

optional - в нем либо лежит T либо ничего не лежит Например используется в функциях, которые могут ничего не возвращать, либо сделать поле в классе - которое либо есть либо нет

variant - позволяет хранить чтото из списка - динамически меняется что в нем лежит

any - можно класть что угодно

Проблема работы с union - что все объекты надо создавать самим и уничтожать самим - строки надо явно уничтожать например. Union не могут наследоваться, но могут быть шаблонными.

Но в C++17 сделали адекватную замену - variant.

Шаблон класса **std::variant** представляет собой безопасный union для типов. Экземпляр std::variant в любой момент времени либо имеет значение одного из своих альтернативных типов, либо, в случае ошибки, не имеет значения (такого состояния трудно достичь).

Как и в случае с union, если std::variant содержит значение некоторого объектного типа T, объектное представление T выделяется непосредственно в объектном представлении самого варианта. Варианту не разрешается выделять дополнительную (динамическую) память.

Варианту не разрешается хранить ссылки, массивы или тип void. Пустые варианты также являются некорректными (вместо них можно использовать std::variant<std::monostate>).

Вариант может содержать один и тот же тип более одного раза, а также содержать различные cv-квалифицированные версии одного и того же типа.

```cpp
#include <variant>

int main() {
    std::variant<int, double, std::string> v = 1;
    std::cout << std::get<int>(v); // 1
    std::cout << std::get<double>(); // exception
    v = "abc";
    std::cout << std::get<std::string>(v);  // "abc"
    v = 5.0;
    std::cout << std::holds_alternative<double>(v); // true
    std::cout << v.index(); // 1
}
```

```cpp
template <size_t N, typename T, typename... Tail>
struct get_index_by_type {
    static const size_t value N;
};

template <size_t N, typename T, typename Head, typename... Tail>
struct get_index_by_type {
    static const size_t value = std::is_same_v<T, Head> ? N:
            get_index_by_type<N + 1, T, Tails...>::value;

};

template <typename... Types>
```

```
class variant;

template <typename T, typename... Types>
struct VariantAlternative {
    // CRTP
    using Derived = variant<Types...>;

    VariantAlternative(const T& value) {
        static_cast<Derived*>(this).storage.put<sizeof...(Types)>(value);
    }

    VariantAlternative(T&. value) {
        static_cast<Derived*>(this).storage.put<sizeof...(Types)>(std::move(
    value));
    }

    void destroy() {
        auto this_ptr = static_cast<Derived*>(this);
        if (get_index_by_type<N, T, Types...?::value == this_ptr->current) {
            // this_ptr->storage.destroy
        }
    }
    // определения надо вынести вниз, после variant
};


template <typename... Types>
class variant: private VariantAlternative<Types, Types...>... {
private:
    template <typename... TTypes>
    union VariadicUnion {};

    temaplte <typename Head, typename... Tails>
    union VariadicUnion<Head, Tail...> {
        Head head;
        VariadicUnion<Tail...> tail;

        tempalte <size_t N, typename T>
        void put(const T& value) {
            if constexpr (N == 0) {
                new (&head) T(value);
            } else {
                tail.put<N - 1>(value);
            }
        }
    };

    VariadicUnion<Types...> storage;
    size_t current = 0;
public:
    using VariantAlternative<Types, Types...>::VariantAlternative...;

    /*
    template <typename T>
    variant(const T& value) {
        // static_assert(T is one of types)
        current = get_index_by_type<0, T, Types...>::value;
        storage.put<get_index_by_type<0, T, Types...>::value>(value);
    }
    */
```

```
73
74      size_t index() const {
75          return current;
76      }
77
78      template <typename T>
79      bool holds_alternative() const {
80          return current == get_index_by_type<0, T, Types...>::value;
81      }
82
83      ~variant {
84          (VariantAlternative<Types, Types...>::destroy(), ...);
85      }
86  };
```

Конструктор можно сделать либо через шаблонный конструктор с проверками. Либо можно отнаследвоаться от такой штуки `VariantAlternative<Types, Types...>...` - то есть это N родителей с типами `<T_i, Types>`. При этом сам VariantAlternative - тоже является variant. И когда мы пишем using... то мы подключаем конустркторы родителя а именно конструкторы VariantAlternative и от нужных типов как раз.

Деструктор тоже можно сделать в классе "родителя";

cppreference:

Destructor:

Example:

```
1  #include <variant>
2  #include <cstdio>
3
4  int main()
5  {
6      struct X { ~X() { puts("X::~X();"); } };
7      struct Y { ~Y() { puts("Y::~Y();"); } };
8
9      {
10         puts("entering block #1");
11         std::variant<X,Y> var;
12         puts("leaving block #1");
13     }
14
15     {
16         puts("entering block #2");
17         std::variant<X,Y> var{ std::in_place_index_t<1>{} };
18         // constructs var(Y)
19         puts("leaving block #2");
20     }
21 }
22
23 /*
24 Output:
25 entering block #1
26 leaving block #1
27 X::~X();
28 entering block #2
29 leaving block #2
30 Y::~Y();
31 */
```

If valueless_by_exception() is true, does nothing. Otherwise, destroys the currently contained value.

This destructor is trivial if std::is_trivially_destructible_v<T_i> is true for all T_i in Types.

Constructors:

Constructs a new variant object.

1) Default constructor. Constructs a variant holding the value-initialized value of the first alternative (index() is zero). This constructor is constexpr if and only if the value initialization of the alternative type T_0 would satisfy the requirements for a constexpr function.

2) Copy constructor. If other is not valueless_by_exception, constructs a variant holding the same alternative as other and direct-initializes the contained value with std::get<other.index()>(other). Otherwise, initializes a valueless_by_exception variant.

This constructor is defined as deleted unless std::is_copy_constructible_v<T_i> is true for all T_i in Types....

3) Move constructor. If other is not valueless_by_exception, constructs a variant holding the same alternative as other and direct-initializes the contained value with std::get<other.index()>(std::move(other)). Otherwise, initializes a valueless_by_exception variant.

4) Converting constructor. Constructs a variant holding the alternative type T_j that would be selected by overload resolution for the expression F(std::forward<T>(t)) if there was an overload of imaginary function F(T_i) for every T_i from Types... in scope at the same time.

```
    std::variant<std::string> v("abc"); // OK
    std::variant<std::string, std::string> w("abc"); // ill-formed
    std::variant<std::string, const char*> x("abc");
    // OK, chooses const char*
    std::variant<std::string, bool> y("abc");
    // OK, chooses string; bool is not a candidate
    std::variant<float, long, double> z = 0;
    // OK, holds long
    // float and double are not candidates
```

И ещё много пунктов (см. оригинал)

extra7. Идиома type erasure. Реализация type erasure на примере класса std::any. Основные методы этого класса. Каким образом работают конструкторы этого класса, копирование и перемещение, присваивание?

```
#include <any>

int main() {
    std::any a = 5;
    std::cout << std::any_cast<int>(a); // 5
    std::cout << std::any_cast<double>(a); // Exception

    a.type(); // std::type_info
    std::cout << a.type().name();

    a = "abcde";
    // std::any_cast<std::string>(a);
    std::any_cast<const char*>(a);
}
```

Implementation of std::any

```
class any {
private:
    void* storage;

    struct Base {
        virtual Base* get_copy();
        virtual ~Base() {}
    };

    template <typename T>
    struct Derived: public Base {
        T value;

        Derived(const T& value): value(value) {}

        Base* get_copy() {
            return new Derived<T>(value);
        }
    };

    Base* storage = nullptr;
public:
    template <typename U>
    any(const U& value): storage(new Derived<U>(value)) {}

    ~any() {
        delete storage;
    }

    any(const any& a): storage(a.storage->get_copy()) {}

    tempalte <typename U>
    any& operator=(const U& value) {
        delete storage;
        storage = new Derived<U>(value);
    }
};
```

Проблема возникает в деструкторе - потому что надо разрушить тип предыдущий, но не понятно как достать его тип. Благодаря такому полиморфизму мы можем подменять один тип на другой, и все будет корректно вызываться. Такой приём называется **идиома Type erasure**.

extra8. Идиома type erasure. Реализация type erasure на примере нестандартного Deleter у класса shared_ptr.

Type Erasure: см. выше.

```
private:
    struct DeleterBase {
    virtual void operator()(T*);
        virtual ~DeleterBase() {}
    };

    template <typename U>
    struct DeleterDerived: public DeleterBase {
        U deleter;
```

```
10          DeleterDerived(const U& deleter):deleter(deleter) {}
11          void operator()(T* ptr) override {
12              deleter(ptr);
13          }
14      };
15
16      DeleterBase* deleter = nullptr;
17  public:
18      ~shared_ptr() {
19              --*counter;
20      }
```

У shared_ptr может быть свой deleter - в случаях если мы хотим не удалять а что-то еще сделать например. А Аллокатор нужен на свои нужды, выделять указатели на контрольные блоки.

extra9. Реализация функции allocate_shared с поддержкой нестандартного аллокатора. Каким образом shared_ptr поддерживает нестандартные аллокаторы, каким образом происходит удаление объекта и контрольного блока в этом случае? Как поддержка нестандартного аллокатора может одновременно сочетаться с поддержкой нестандартного Deleter?

Постановка проблемы: мы хотим, чтобы `shared_ptr` работал с кастомным аллокатором (да и вообще, обращение к new - плохой кодстайл).

Поэтому, нам нужно научиться этот аллокатор где-то хранить. Очевидно, это нельзя делать внутри полей. Идея - та же что и с нестандартным делитером - нам нужно использовать type erasure для аллокатора. Но есть небольшая проблема - мы этот аллокатор должны хранить, и одновременно им освобождать выделенную под него память. Кажется, это решается следующим способом: сначала делаем копию аллокатора, а потом этим новым аллокатором делаем `deallocate` всего чего нужно.

```
1
2  struct BaseAllocator {
3    virtual void dealloc(void *ptr) = 0;
4    virtual bool TAllocated() = 0;
5    virtual ~BaseAllocator() = default;
6  };
7
8  template<typename T, typename U = std::allocator<char>>
9  struct AllocatorWithNotCstyle : BaseAllocator {
10   U allocator;
11   AllocatorWithNotCstyle(U allocator) : allocator(allocator) {}
12   void dealloc(void *ptr) override {
13     using right_allocator_type = typename std::allocator_traits<U>::template
       rebind_alloc<char>;
14     right_allocator_type right_allocator = allocator;
15     //тут происходит копия аллокатора
16
17     using right_traits = std::allocator_traits<right_allocator_type>;
18
19     //кажется, тут еще должен быть destroy аллокатора allocator, а именно:
20     //U new = allocator; - копируем аллокатор
21     //typename std::allocator_traits<U>::destroy(new, &allocator);
22
23     right_traits::deallocate(right_allocator,
24                              reinterpret_cast<char *>(ptr),
25                              sizeof(T) + sizeof(BaseDeleter) + sizeof(
       BaseAllocator) + 2 * sizeof(size_t));
26   }
27   bool TAllocated() override {
28     return 1;
29   }
30 };
31
32 template<typename T, typename U = std::allocator<char>>
33 struct AllocatorWithCStyle : BaseAllocator {
34   U allocator;
35   AllocatorWithCStyle(U allocator) : allocator(allocator) {}
```

```
  void dealloc(void *ptr) override {
    using right_allocator_type = typename std::allocator_traits<U>::template
     rebind_alloc<char>;
    right_allocator_type right_allocator = allocator;

    //тут нужно добавить то же что и выше

    using right_traits = std::allocator_traits<right_allocator_type>;
    right_traits::deallocate(right_allocator,
                             reinterpret_cast<char *>(ptr),
                             sizeof(BaseDeleter) + sizeof(BaseAllocator) + 2
    * sizeof(size_t));
  }
  bool TAllocated() override {
    return 0;
  }
};
```

В этом коде, к сожалению, отсутствует Control Block, но что уж есть.

```
template<typename T, typename Allocator, typename... Args>
SharedPtr<T> allocateShared(const Allocator &alloc, Args &&... args) {
  using charAllocatorType = typename std::allocator_traits<Allocator>::
   template rebind_alloc<char>;
  using TAllocatorType = typename std::allocator_traits<Allocator>::template
    rebind_alloc<T>;
  charAllocatorType charAllocator = alloc;
  TAllocatorType TAllocator = alloc;
  char *ControlBlock = std::allocator_traits<charAllocatorType>::allocate(
   charAllocator,

   sizeof(T) + sizeof(BaseDeleter)

     + sizeof(BaseAllocator)

     + 2 * sizeof(size_t));
  std::allocator_traits<TAllocatorType>::construct(TAllocator,
                                                   reinterpret_cast<T *>(
   ControlBlock),
                                                   std::forward<Args>(args)
   ...);
  auto deleter_ = reinterpret_cast<BaseDeleter *>(ControlBlock + sizeof(T));
  auto allocator_ = reinterpret_cast<BaseAllocator *>(ControlBlock + sizeof(
   T) + sizeof(BaseDeleter));
  new(deleter_) DeleterWithAllocator<T, Allocator>(TAllocator);
  new(allocator_) AllocatorWithNotCstyle<T, Allocator>(TAllocator);
  return SharedPtr<T>(ControlBlock);
}
```

Тут просто происходит следущее: для аллокатора мы применям ровно ту же технику, что и для делитера в предыдущем билете. Единственное отличие заключается в том, что мы должны этим же аллокатором себя почистить, для этого просто делаем копию

10.10 Функция std::invoke, ее предназначение и идея реализации. Зачем нужна эта функция? Реализация метафункции std::invocable<F, Args...>

Invoke - вызвать функцию. Пример использования:

```
int main() {
    std::function<bool(int, int)> compare = [](int x, int y) { return x < y;
    };
    cout << std::invoke(compare, 1, 2);
    //output 1;
}
```

std::invoke нужен чтобы единым образом вызвать функторы (в т.ч. лямбды), указатели на функции и указатели на функции-члены классов. Последние имеют специфический синтаксис вызова не совпадающий с синтаксисом обычных функторов:

```
(obj->*mem_fn_ptr)( args... );
```

Если вам нужно реализовать поддержку всех callable объектов, то проще передать их одним пакетом в invoke, чем писать отдельную шаблонную реализацию для каждого случая. Такой синтаксический сахар.

Реализация метафункции std::invocable<F, Args...>

```
namespace detail {
template<class>
constexpr bool is_reference_wrapper_v = false;
template<class U>
constexpr bool is_reference_wrapper_v<std::reference_wrapper<U>> = true;

template<class C, class Pointed, class T1, class... Args>
constexpr decltype(auto) invoke_memptr(Pointed C::* f, T1&& t1, Args&&...
    args)
{
    if constexpr (std::is_function_v<Pointed>) {
        if constexpr (std::is_base_of_v<C, std::decay_t<T1>>)
            return (std::forward<T1>(t1).*f)(std::forward<Args>(args)...);
        else if constexpr (is_reference_wrapper_v<std::decay_t<T1>>)
            return (t1.get().*f)(std::forward<Args>(args)...);
        else
            return ((*std::forward<T1>(t1)).*f)(std::forward<Args>(args)...)
    ;
    } else {
        static_assert(std::is_object_v<Pointed> && sizeof...(args) == 0);
        if constexpr (std::is_base_of_v<C, std::decay_t<T1>>)
            return std::forward<T1>(t1).*f;
        else if constexpr (is_reference_wrapper_v<std::decay_t<T1>>)
            return t1.get().*f;
        else
            return (*std::forward<T1>(t1)).*f;
    }
}
} // namespace detail

template<class F, class... Args>
constexpr std::invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
    noexcept(std::is_nothrow_invocable_v<F, Args...>)
{
```

```
33      if constexpr (std::is_member_pointer_v<std::decay_t<F>>)
34          return detail::invoke_memptr(f, std::forward<Args>(args)...);
35      else
36          return std::forward<F>(f)(std::forward<Args>(args)...);
37  }
```

Идея реализаций: расписываем вручную все варианты, и в зависимости от этого выбираем нужную сигнатуру

extra11. Реализуйте метафункцию is_class<T>, позволяющую проверить, является ли тип T классом или структурой. Что можно сказать о реализации is_union<T>?

Реализация is_class<T>:

```
namespace detail {
  template <class T> char test(int T::*);
  struct two { char c[2]; };
  template <class T> two test(...);
}

template <class T>
struct is_class : std::integral_constant<bool, sizeof(detail::test<T>(0))==1
    && !std::is_union<T>::value> {};
```

Пример использования:

```
struct A {};

class B {};

enum class C {};

int main()
{
  std::cout << std::boolalpha;
  std::cout << std::is_class<A>::value << '\n';
  std::cout << std::is_class<B>::value << '\n';
  std::cout << std::is_class<C>::value << '\n';
  std::cout << std::is_class<int>::value << '\n';
}
```

Основные мысли:
1. Используем SFINAE и радуемся жизни. тык почитать поподробнее
2. Class и struct вообще в c++ неразличимы.

extra12. Концепты в C++20. Синтаксис определения концептов, ключевое слово requires. Использование концептов в качестве параметров шаблонов. Определение концептов ForwardIterator, BidirectionalIterator, RandomAccessIterator. Решение проблемы с реализацией функции std::advance с помощью концептов.

Проблемы обобщённого программирования на C++:

1. *Ошибки при использовании шаблонов вылезают непонятно где, они трудны для восприятия.*

2. *Трудно писать разные реализации одной шаблонной функции для разных категорий типов.* Пусть, я хочу написать функцию, которая проверяет, что два числа достаточно близки друг к другу. Для целых чисел достаточно проверить, что числа равны между собой, для чисел с плавающей точкой — то, что разность меньше некоторого ε.

Задачу можно решить хаком SFINAE, написав две функции. Хак использует `std::enable_if`. Это специальный шаблон в стандартной библиотеке, который содержит ошибку в случае если условие не выполнено. При инстанцировании шаблона компилятор отбрасывает декларации с ошибкой:

```
#include

template
T Abs(T x) {
    return x >= 0 ? x : -x;
}

// вариант для чисел с плавающей точкой
template
std::enable_if_t, bool>
AreClose(T a, T b) {
    return Abs(a - b) < static_cast(0.000001);
}

// вариант для других объектов
template
std::enable_if_t, bool>
AreClose(T a, T b) {
    return a == b;
}
```

Обе проблемы легко решить, если добавить в язык всего одну возможность — накладывать **ограничения на шаблонные параметры**. Например, требовать, чтобы шаблонный параметр был контейнером или объектом, поддерживающим сравнение. Это и есть концепт.

Перепишем нашу плавающую точку при помощи концептов:

```
#include

template
T Abs(T x) {
    return x >= 0 ? x : -x;
}

// вариант для чисел с плавающей точкой
template
```

```
requires(std::is_floating_point_v)
bool AreClose(T a, T b) {
    return Abs(a - b) < static_cast(0.000001);
}

// вариант для других объектов
template
bool AreClose(T a, T b) {
    return a == b;
}
```

Аналогично можно расписать функцию печати контейнера:

```
#include
#include

template
concept HasBeginEnd =
    requires(T a) {
        a.begin();
        a.end();
    };

template
void Print(std::ostream& out, const T& v) {
    for (const auto& elem : v) {
        out << elem << std::endl;
    }
}

template
void Print(std::ostream& out, const T& v) {
    out << v;
}
```

Концепт — это имя для ограничения.

Мы свели его к другому понятию, определение которого уже содержательно, но может показаться странным:

Ограничение — это шаблонное булево выражение.

Грубо говоря, приведённые выше условия «быть итератором» или «являться числом с плавающей точкой» — это и есть ограничения. Вся суть нововведения заключается именно в ограничениях, а концепт — лишь способ на них ссылаться.

Для ограничений доступны булевы операции и комбинации других ограничений; в ограничениях можно использовать выражения и даже вызывать функции. Но функции должны быть constexpr — они вычисляются на этапе компиляции:

```
template
concept Integral = std::is_integral::value;

template
concept SignedIntegral = Integral &&
                         std::is_signed::value;
template
concept UnsignedIntegral = Integral &&
                           !SignedIntegral;

template
constexpr bool get_value() { return T::value; }

template
    requires (sizeof(T) > 1 && get_value())
```

```
void f(T); // 1

void f(int); // 2

void g() {
    f('A'); // вызывает 2.
```

Для ограничений есть отличная возможность: проверка корректности выражения — того, что оно компилируется без ошибок. Посмотрите на ограничение Addable. В скобках написано a + b. Условия ограничения выполняются тогда, когда значения a и b типа T допускают такую запись, то есть T имеет определённую операцию сложения:

```
template
concept Addable =
requires (T a, T b) {
    a + b;
};
```

Ограничение может требовать не только корректность выражения, но и чтобы тип его значения чему-то соответствовал. Здесь мы записываем:

выражение в фигурных скобках,

->,

другое ограничение.

```
template concept C1 =
requires(T x) {
    {x + 1} -> std::same_as;
};
```

Фулл

Концепты для итераторов:

```
template<class I>
  concept forward_iterator =
    std::input_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::forward_iterator_tag> &&
    std::incrementable<I> &&
    std::sentinel_for<I, I>;
```

фулл с пояснениями - 1

```
template<class I>
  concept bidirectional_iterator =
    std::forward_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::bidirectional_iterator_tag>
   &&
    requires(I i) {
      { --i } -> std::same_as<I&>;
      { i-- } -> std::same_as<I>;
    };
```

фулл с пояснениями - 2

```
template<class I>
  concept random_access_iterator =
    std::bidirectional_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::random_access_iterator_tag>
   &&
    std::totally_ordered<I> &&
    std::sized_sentinel_for<I, I> &&
    requires(I i, const I j, const std::iter_difference_t<I> n) {
      { i += n } -> std::same_as<I&>;
      { j +  n } -> std::same_as<I>;
```

```
      { n +  j } -> std::same_as<I>;
      { i -= n } -> std::same_as<I&>;
      { j -  n } -> std::same_as<I>;
      {  j[n]  } -> std::same_as<std::iter_reference_t<I>>;
    };
```

фулл с пояснениями - 3

Беды с std::advance

std::advance(iter, n) puts its iterator iter n position further. Depending on the iterator, the implementation can use pointer arithmetic or just go n times further. In the first case, the execution time is constant; in the second case, the execution time depends on the stepsize n. Thanks to concepts, you can overload std::advance on the iterator category.

```
template<InputIterator I>
void advance(I& iter, int n){...}

template<BidirectionalIterator I>
void advance(I& iter, int n){...}

template<RandomAccessIterator I>
void advance(I& iter, int n){...}

// usage

std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vecIt = vec.begin();
std::advance(vecIt, 5);       // RandomAccessIterator

std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lstIt = lst.begin();
std::advance(lstIt, 5);       // BidirectionalIterator

std::forward_list<int> forw{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto forwIt = forw.begin();
std::advance(forwIt, 5);      // InputIterator
```

фулл