Введение в ООП

Анастасия Вознюк, Полина Чубенко, Ксения Куринова

Январь - Июнь 2021

Я, прочитав над входом, в вышине,
Такие знаки сумрачного цвета,
Сказал: «Учитель, смысл их страшен мне»

А. Данте

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

Е. Замятин

Язык C++ неисчерпаемо богат и все обогащается с быстротой поражающей.

М. Горький

Содержание

1 Базовые понятия

1.1 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**

Параметры компилятора:

- **-Wall** Включает все предупреждения, в том числе предупреждения, отключенные по умолчанию.

- **-Wextra** Некоторые предупреждения не выводятся даже при использовании параметра выше, поэтому используют этот параметр.
- **-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:

```
1    // assignment in a conditional statement
2    int x=3, y=4;
3    if(x=y) {}
4
5    // implicit type conversion
6    int n = 10;
7    for(size_t i = 0; i < n; ++i){}
```

1.2 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)
   - × int const x = 5;
   - × 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){}

1.3 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.

1.4 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:

- × Для **static_cast** преобразование указателя на родительский класс к указателю на дочерний класс. Объект по указателю обязан быть правильного дочернего класса, иначе это undefined behaviour.
- × Битые ссылки.
- × **++x = x++;** или **f(x=y, x=3);** — порядок вычисления аргументов оператора и функций не определён. (until C++17)
- × **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!
- × Выход за границы **C-style массива**.

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

1.5 Интересное UB с параметром -O2

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

```
1    for(int i = 0; i < 300; i++){
2        cout << i << " " << i * 12345678 << endl;
3    }
```

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.

1.6 Unspecified behavior

Неопределенное поведение подразумевает использование неопределенного значения или другого поведения, когда настоящий Международный стандарт предоставляет две или более возможностей и не налагает никаких дополнительных требований на то, какое поведение выбирается в том или другом случае

× порядок, в котором вычисляются аргументы функции или сами функции, т.е.

```
1    std::cout << f() + g() * g();
2    f(g(), h());
```

1.7 Выражения и операторы

1. **Идентификаторы** — любая последовательность латинских букв, цифр и знака "_", не начинающаяся с цифры. Они не могут совпадать с ключевыми словами (new, delete, class, int, if, true, etc)

2. **Литералы** — последовательность символов, интерпретируемая как константное значение какого-то типа (1, 'a', "abc", 0.5, true, nullptr, etc)

3. **Операторы** — это, можно сказать, функции со специальными именами (=, +, <, [], (), etc)

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

5. **Тернарный оператор** (?:). "Условие" ? "выражение, если true" : "выражение, если false"

6. **Оператор "запятая"** (,). Вычисляет то, что слева, затем вычисляет то, что справа и возвращает то, что справа. (Имеет самый низкий приоритет)

7. **Унарная "звёздочка"** (*). Разыменование

8. **Унарный "амперсанд"** (&). Взятие адреса

9. **Оператор "точка"/"стрелочка"**. Доступ к полю/методу класса (соответственно через объект класса/указатель на объект)

10. **Двойное двоеточие**. Переход в другую область видимости (std::cout, ::operator new)

11. **Префиксный/постфиксный инкремент**. Префиксный увеличивает на единицу и возвращает ссылку на уже измененный объект. Постфиксный увеличивает на единицу и возвращает копию старого объекта.

12. **Бинарный амперсанд**. Побитовое И

13. **Бинарный двойной амперсанд**. Логическое И
14. **Оператор присваивания**. Присваивает значение (копированием или перемещением)
15. **Оператор составного присваивания**. Легче пример: $a+=5 \Leftrightarrow a=a+5$. Только во втором случае создается лишняя копия
16. **Оператор** $<<$ ($>>$). В заивисмости от контекста это либо побитовое смещение влево (вправо), либо это оператор ввода(вывода) в(из) поток(а).

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=`	For relational operators < and ≤ respectively	
	`>` `>=`	For relational operators > and ≥ respectively	
10	`==` `!=`	For equality operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

Пример: int x = 0; ++x = x++. Выводит ноль или UB в зависимости от стандарта.
Пример: ++x++ даёт CE, так как ++(x++).
Замечание: приоритет оператора и порядок вычисления – разные вещи.

1.8 Основные типы и операции над ними

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 / 255
целочисленные типы данных		
short	2	-32 768 / 32 767
unsigned short	2	0 / 65 535
int	4	-2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long long	8	-2^{63} / $2^{63}-1$
unsigned long long	8	0 / $2^{64}-1$
вещественные типы данных		
float	4	-2 147 483 648.0 / 2 147 483 647.0
double	8	-2^{63} / $2^{63}-1$
long double	10	.....
символьные типы данных		
char	1	-128 / 127
unsighned char	1	0 / 255

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 в сумме очень большое число.

Note: size_t беззнаковый целый тип данных, возвращаемый оператором sizeof, определен в заголовочном файле <cstring>

1.9 Виды памяти, утечка памяти, сборка мусора

1. **Стековая память** — память, в которой находятся все локальные переменные функций. Её у нас условно где-то несколько мегабайт
2. **Динамическая память** — некоторая память процесса, которая выдаётся нам по запросу (через new/malloc/calloc). Её гораздо больше, чем стековой (столько, сколько есть у всей системы). Обращение к ней происходит через указатели
3. **Stack overflow** — переполнение стековой памяти. (Слишком глубокая рекурсия или обычный массив большого размера). Это RE

4. **Утечка памяти** — это когда вы выделили динамическую память, но не вернули её системе по окончании её использования (или при каком-то аварийном завершении).

5. **Сборка мусора** — механизм автоматического управления памяти, когда ты не заботишься о том, чтобы вызвать delete, система всё делает за тебя. Есть в Джаве (Java) и Питоне (Python), в плюсах (C++) отсутствует (хотя умные указатели — вполне неплохая альтернатива)

1.10 Ссылки и указатели, битые ссылки

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.

Nullptr - ключевое слово, введенное в C++11 для описания константы нулевого указателя. Данная константа имеет тип std::nullptr_t. Nullptr является константой r-value.
Что будет, если разыменовать nullptr? UB. Не поддерживает вывод.

Ссылка — особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Ссылка — это новое название для уже существующей переменной.
Различия

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

```
1     // this WILL NOT compile
2     int a = 0;
3     int&* b = a;
4
5     // but this WILL
6     int a = 0;
7     int& b = a;
8     int* pb = &b; //pointer to a
9
10    // and this WILL
11    int* a = new int;
12    int*& b = a; //reference to pointer - change b changes a
```

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

- ```
        int main() {
        int x;
        const int cx = 9;
        const int& cy = x; //constant ref
        int& const pcy = cx; //WRONG, ref already const
    }
  ```

Замечание: Константную ссылку можно создать от любого объекта, но неконстантную ссылку от константного объекта — нельзя.

Плюсы и минусы использования того и другого:

- ссылки лучше использовать когда нежелательно или не планируется изменение связи ссылка $\longrightarrow$ объект
- указатель лучше использовать, когда возможны следующие моменты в течении жизни ссылки:
  - ссылка не указывает ни на какой объект;
  - ссылка указаывает на разные объекты в течении своего времени жизни.

Битая ссылка — 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
    }
```

Дополнение: Ссылки можно делать полями классов, причем инициализировать их можно как на месте (since C++11):
```

```
struct C {
    int field = 0;
    int& field_alias = field;
};
// OR
struct C {
    C(int& x) : x(x) {}
    int& x;
};
```

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

1.11 Объявление, определение и области видимости

Что можно объявлять?

1. Переменные
2. Функции
3. Свои собственные структуры
4. Классы
5. Union
6. Namespace
7. Объявление alias (псевдонимы)

1.11.1 Разница между определением и объявлением

Definition это ситуация, в которой пользователь не только объявил, но и определил. В случае переменных, согласно стандарту, разницы нет, но уже в случае функций видна разница:

```
int f();
int g() {
    f();
}
int f() {
    g();
}
```

One definition rule: каждая функция и переменная должна быть определена ровно один раз (ничего не говорит про declaration!).

1.12 Области видимости

Scope: При объявлении программного элемента, такого как класс, функция или переменная, его имя может быть "видимым"и использоваться в определенных частях программы. Контекст, в котором отображается имя, называется его областью действия. Примеры:

1. Глобальная область

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

1.12.1 Скрытие имен с глобальной областью видимости

Можно скрыть имена с глобальной областью, явно объявляя одно и то же имя в области видимости блока. Однако доступ к именам глобальных областей можно получить с помощью оператора разрешения области (::).

```
int i = 7;   // i has global scope, outside all blocks
int main() {
    int i = 5;   // i has block scope, hides i at global scope
    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "Global-scoped i has the value: " << ::i << "\n";
}
//Output:
//Block-scoped i has the value: 5
//Global-scoped i has the value: 7
```

В терминологии C++ есть понятие unqualified-id и qualified-id. Если есть префикс (пример: std), то id первого типа, иначе – второго.

1.13 Управляющие конструкции

Можно писать только в теле функции. Основные:

1.
```
if (bool condition) {
    statement;
} else {
    ...;
}
```

```
switch(expr) {
case 1:
    statements;
    break;
...
default;
}
```

3.
```
for (decl or expr; bool-expr; expr) {
    statements;
    continue;
}
```

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

1.14 Массивы

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.

1.15 Функции

1.15.1 Перегрузка функций

Возможность объявлять несколько функций с одинаковыми названиями, но разными типами принимаемых значений.

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
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
5. ellipsis cconvertion

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

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);
11  }
```

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

Замечание: компилятор не смотрит при этом на тип возвращаемого значения.

1.15.2 Аргументы по умолчанию

```
1     void f(int a, int b = 2) {
2         cout << 2;
3     }
4     int main () {
5         f(5);
6         f(5, 9);
7     }
```

Рассмотрим ограничения:

1. Аргументы по умолчанию всегда стоят в конце
2. Наличие аргумента по умолчанию не является основанием для перегрузки

1.15.3 Указатели на функции

void (*ptr)(types...) = function_name
Пример использования: std::sort имеет в качестве параметра компаратор. Можно передать в неё указатель на функцию

1.16 Константы

Константным является тип, который нельзя менять. Другое определение – константным является тип, над которым нельзя выполнять неконстантные операции (например, сортировку).
Рассмотрим определение константной ссылки:

```
1     int main() {
2         int x;
3         const int cx = 9;
4         const int& cy = x; //
5         int& const pcy = cx; //                    :

6     }
```

Пример использования ссылок: передача по ссылке, а не по значению, в функцию, с целью избегания ненужных копий.

Рассмотрим код:

```
1
2     int find(const string text; const string str) {
3         //...
4         return ans;
5     }
6
7     int main() {
8        find("abc", "def");
9        const int& r = 0; // OK
10       //
11       int& pr = 0;// WRONG
12    }
```

1.16.1 lifetime expansion

Не всегда выход ссылки из области видимости означает уничтожение объекта. Если объект был изначально объявлен как ссылка:

```
1    int main() {
2        {
3            const vector<int>& v = {1, 2, 3};
4        }
5    }
```

то этот объект уничтожается не сразу, а живёт до тех пор, пока ассоциированное с ним впервые имя не выйдет из области видимости. Работает только для константных типов.

1.17 Приведение типов

1.17.1 Static cast

Создание новой сущности из старой. Работает на этапе компиляции. Берёт объект старого типа и возвращает нового:

```
1        int main() {
2            int x = 0;
3            double d = static_cast<double>(x);
4        }
```

Работает в том числе и с пользовательскими правилами приведения типа.

Запрещённые заклинания:

1.17.2 Reinterpret cast

Позволяет трактовать байты одного типа как байты другого типа. Бывает в двух формах:
1. От указателя к указателю. Если есть указатель на сущность A, мы умеем воспринимать это как указатель, что указывает на сущность B.

```
1         int main() {
2            int x = 0;
3            double d = reinterpret_cast<double>(x);
4            cout << *reinterpret_cast<double>(&x); //OK
5            cout << *static_cast<double>(&x); //WRONG
6        }
```

Как это работает, если int занимает меньше по памяти, чем double? ну, поэтому это и UB.
2. Более жёсткий вариант: реинтерпрет к ссылке. Смотрим на байты объекта и интерпретируем их как байты другого типа.

1.17.3 Const cast

Рассмотрим пример, который является UB:

```
1    void f(const int *ptr) {
2        // *ptr = 42;
3        // ptr is const.
4        *const_cast<int *>(ptr) = 42; // OK
5    }
```

```
6
7       int main() {
8           int foo = 0; //      .
9           // const int foo = 0; WRONG, changing const
10          f(&foo);
11      }
```

1.17.4 C-style cast

Работает примерно в таком порядке:

1. Пытается в const cast
2. Пытается в static cast
3. Пытается в const + static
4. Пытается в reinterpret + const
5. Иначе CE

2 Вступление. Идея ООП

Программирование состоит из создания объектов некоторого типа - экземпляров этого типа. Каждый тип позволят проводить над собой некоторые операции. Тогда программирование в парадигме ООП сводится к тому, что мы определяем какие-то примитивные типы, свои типы с какими-то операциями, далее вся программа сводится к созданию обхектов каких-то типов и выполнение операций над ними, а затем - уничтожение этих объектов.

2.1 Создание классов

Переход к от обычного структурного программирования к ООП начинается тогда когда у нас появляется возможность определять собственные типы.

1. Создание собственного типа

   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.

2. Поля и методы:

   У классов и структур есть свои поля и методы.

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

   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,

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.

```
1   class C {
2   private:
3       int s = 0;
4       std::string str;
5       double d = 0.0;
6   public:
7       void add_and_print(int a);
8       void add_and_print(double a) {
9           std::cout << d + a;
10      }
11  };
12  void C::add_and_print(int s) {
13      std::cout << C::s + s;
14      //std::cout << this->s + s;
15  }
16  int main() {
17      C c;
18      c.add_and_print(1);
19  }
```

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.

2.2 Три волшебных слова ООП

Инкапсуляция, наследование и полиморфизм - главные принципы, на которых основано ООП

2.3 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. Пример из жизни : ограниченное число кнопок на микроволновке, каждая из которая делает что-то определенное, и у нас нет доступа к ее полному функционалу, чтобы случайно что-то не сломать.

Модификаторы доступа:

1. private - к этим полям/методам нельзя получить доступ извне. Пусть создали объект в другой области видимости - например в функции main - нельзя будет обратиться к private-полям этого объекта из main()

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.

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.

Конструкторы:

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.

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

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

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

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

```
class String{
    String(...): String(...) {
    //smth that is need to be done only by second constructor//
```

```
    }
};
```

```
class String{
private:
    char* str = nullptr;
    size_t sz = 0;
public:
    String() {
    }
    String(size_t sz, char s = '\0'){
    \\...\\
    }
};
```

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

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

Аналогично смотрится сначала перегрузка, и только потом проверяется можно ли вызывать этот конструктор или нет.

Конструкторы с помощью списка инициализации:

```
class String{
    String(std::initializer_list<char> lst){
    size = lst.size();
    str = new char[size];
    std::copy(lst.begin(), lst.end(), str);
    }
};

int main() {
String s = {'a', 'b', 'c'};
}
```

Правило: Среди конструкторов, компилятор в первую очередь пытается вызывать со списком инициализаторов (даже при условии что придется делать приведение типов).

Деструкторы:

Деструктор - метод, который вызывается непосредственно перед тем, как объект будет уничтожен. Как и любой другой метод, его можно определять вне класса. Деструктор генерируется по умолчанию, если он тривиальный - можно не писать default, как это было с конструкторами.

```
~String();
```

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 .

```
    ~String() {
        delete[] str;
    }
```

2.4 Конструктор копирования, оператор присваивания и правило трех

Для большинства объектов хочется уметь присваивать одному объекту класса другой объект этого же класса. Это реализуется с помощью **конструктора копирования**. Если мы не напишем конструктор копирования, то компилятор реализует его сам. 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.

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

Правило трех: Если в классе потребовалось реализовать нетривильный деструктор, или нетривиальный конструктор копирования или нетриваильный оператор присваивания, то в классе нужно реализовать все три.

Дефолтный оператор присваивания тоже работает по принципу shallow copy

```
    String& operator= (const String& s) = default;
    //or
    String& operator= (const String& s){
        if (this == &s) return *this; //self-assignment
    };
```

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).

2.5 Member initializer lists - список инициализации членов

Способ инициализировать поля до их входа в конструктор. Если поля уже проинциализированы в самом классе, то при входе в метод комплиятор создаст и проинициализирует их - получаем лишнюю работу, если далее конструктор класса изменит все поля.

```
    class String{
    private:
        size_t size =0;
        char* str;
    public:
        String (size_t sz, char c) : size(s.size), str = new char[size]{
        }
    }
```

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 MIL и делегирование конструкторов - CE.

3 Перегрузка операторов

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

3.1 Перегрузка арифметических операторов

В метод класса передаем только второй операнд, так как под левым операндом подразумевается *this.

```
1     struct Complex{
2         double re = 0.0;
3         double im = 0.0;
4
5         Complex& operator +=(const Complex& z) {
6             re += z.re;
7             in += z.im;
8             return *this;
9         }
10        Complex operator +(const Complex& z) {
11            Complex copy = *this;
12            copy += z;
13            return copy;
14        }
15
16    }
```

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

```
1     Complex c(2.0);
2        += 1.0;
```

или при вызове

```
1     Complex c(2.0);
2        + 1.0;
```

все сработает хорошо, компилятор сделает неявное преобразование double к Complex, однако следующий вызов

```
1     Complex c(2.0);
2     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:

```
1     struct Complex{
2         double re = 0.0;
3         double im = 0.0;
4
5         Complex (const Complex&){
6         }
7
8         Complex& operator +=(const Complex& z) {
9             re += z.re;
10            in += z.im;
11            return *this;
12        }
13    }
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    }
```

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 (23) 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 не в закомментированном коде это оптимизация не буедт вызвана, хотя и возвращается объект, созданный в методе - компилятору не очевидно, что это тот же самый объект.

3.2 Перегрузка операторов ввода и вывода

Не может быть членом класса, так как его левый операнд - поток. Возвращаемый тип оператора ввода и вывода - ссылка на поток (чтобы можно было писать cout « a « b).

3.3 Еще немного о Copy Elision и RVO

Если метод явно возвращает объект, то с C++17 гарантируется, что произойдет Copy Elision

```
    struct C {
        C() {}
        C(const C&) { std::cout << "A copy was made.\n"; }
    };

    C f() {
        return C(); //Definitely performs copy elision, regular RVO
    }
    C g() {
        C c;
        return c; //Maybe performs copy elision, named RVO
    }

    void foo(C c); //RVO, as a temporary object passed by value

    int main() {
    std::cout << "Hello World!\n";
    C obj = f(); //Copy constructor isn't called
    }
    foo(C());
```

3.4 Перегрузка префиксного и постфиксного инкремента

Чтобы отличить префиксный инкремент от постфиксного, при реализации постфиксного нужно передавать тип int в качестве параметра (при вызове самого оператора - не нужно, это костыль для того чтобы различать в классе два метода). Кроме того, префиксный инкремент возвращает ссылку на объект - так как результат должен быть l-value, а постфиксный инкремент - возвращает копию объекта.

```
    BigInteger& operator++{
        *this += 1;
        return *this;
    }

    BigInteger& operator++(int){
        BigInteger copy = *this;
        ++*this;
        return copy;
    }
```

3.5 Перегрузка операторов сравнения

Сначала перегружаем оператор $<$, и через него выражаем все остальные операторы сравнения.

```
    bool operator< (const BigInteger& a, const BigInteger& b);
    bool opeartor> (const BigInteger& a, const BigInteger& b){
        return b < a;
```

```
4       }
5       bool operator== (const BigInteger& a, const BigInteger& b){
6           return !(b < a || a < b);
7       }
```

4 Различные методы классов

4.1 Константные и неконстантые методы, перегрузка []

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.

Оператор [] нужно перегружать для константных и неконстантых строк отдельно, так как во втором случае мы должны возвращать по ссылке для возвращения l-value, чтобы можно было изменять строку по значению, а в первом случае - по константной ссылке, иначе можно было бы изменять константную строку.

```
1     String{
2     char* str = nullptr; //will become pointer to const, not const pointer
          thus such method might change the const string
3     size_t size = 0;
4
5     const char& operator[] (size_t ind) const{ //or return char by value
6         return str[index];
7         }
8     }
9
10    char& operator[] (size_t ind){
11        return str[index];
12        }
13    }
```

4.2 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.

```
1     class A{
2         int s;
3         void f(int);
4     };
5     class B{
6         int t;
7     }
8     class String{
9         char* str = nullptr;
10        size_t size = 0;
11
12        friend void f(int);
13        friend void A::g(int);
14        friend class B;
15
16    };
```

4.3 Статические поля и методы

Статические поля хранятся в статической памяти, создаются при запуске программы, существуют до конца программы и являются общими для всех объектов этого класса - в единственном экземпляре на все объекты. Пример статического поля - сколько существует объектов данного класса. Замечательно свойство - к этому методу класса можно обратиться напрямую, не создавая объектов этого класса. Если статический метод публичный, то для вызова его извне класса надо писать:

```
1     class A {
2     public:
3         static int x;
4     };
5
6     int main(){
7         A::x = 1;
8         A a;
9         A aa;
10        aa.x = 2;
11        std::cout << a.x << aa.x << A::x; //all three will become 2
12    }
```

Если статическое поле не является константой целочисленного типа, его надо инициализировать вне класса.

Из статических методов нельзя обращаться к нестатическим полям или методам. Если статический метод публичный, то для вызова его извне класса надо писать:

```
1     class A{
2     public:
```

```
3         static void method();
4     };
5
6     int main(){
7         A::method;
8     }
```

Пример использования статических методов: singleton - класс, который гарантирует что объект всегда будет в единственном экземпляре.

4.4 Перегрузка приведения типов. Explicit

Конверсия $int \to UserId$ выполняется конструктором, а $UserId \to int$ - оператором приведения типа. Для оператора приведения типов не пишется возвращаемое значение. В следующем коде получаем, что возможно сложение UserId и GroupId, что бессмысленно. Но еще может происходить неявное преобразование, которое не отследить компилятором.

```
1     class UserId{
2     private:
3         int id = 0;
4     public:
5         UserId(int id): id(id){};
6         operator int(){
7             return id;
8         }
9     };
10
11    class GroupId{
12    private:
13        int id = 0;
14    public:
15        GroupId(int id): id(id){};
16        operator int(){
17            return id;
18        }
19    };
20
21    int main(){
22        UserId id = 5;
23        std::cout << id + 5; //10
24    }
```

Можно попросить компилятор запретить неявное преобразование одного типа в другой(в этом смысл разбивать на различные классы) с помощью ключевого слова **explicit**. Применяется только к конструктору и оператору приведения типов и запрещает неявные вызовы этих операторов - теперь только явно можно вызывать преобразование к int и конструктор от int. Пишем:

```
1     class UserId{
2         explicit UserId(int id): id(id){};
3         explicit operator int(){
4             return id;
5         }
```

```
6     };
7     class GroupId{
8         explicit GroupId(int id): id(id){};
9         explicit operator int(){
10            return id;
11        }
12    };
```

Теперь вызов UserId id = 5 некорректен (нужно UserId id(5)). Рекомендуется писать explicit ко всем конструкторам от одного аргумента.

Конверсия bool в int в различных условиях (if, while) не считается неявной конверсией (non-explicit). Такая конверсия игнорирует explicit и называется контекстуальной конверсией - **contextual conversion**.

4.5 Перегрузка литеральных суффиксов

Пусть хотим чтобы 5_uid считалось бы константой типа UserId. В параметрах перегруженного оператора можно указывать только переменную типа char или const char* или unsigned long long.

```
1     class UserId{
2         explicit UserId(int id): id(id){};
3         explicit operator int(){
4             return id;
5         }
6     };
7
8     UserId operator ""_uid (unsigned long long x) {
9         return UserId(x);
10    }
11
12    int main(){
13    UserId = 5_uid;
14    }
```

4.6 Перегрузка оператора круглые скобки

Переопределив оператор(), можно вопринимать объекты как функции от некоторого конкретного набора параметров, такие объекты называются функциональными объектами или функторами Нужны для создания компараторов и других функторов предикатов. Преимущества перед функциями появляются когда мы например хотим реализовать set со своим способом упорядочивания, а там требуется тип "сравнивателя". Кроме того, передавать функцию/указатель на функцию считается C-Style.

```
1     struct GreaterThanZero{
2         bool operator()(int t){
3             return t > 0;
4         }
5     };
6
7     struct comp {
```

```
8           bool operator() (int left, int right){
9               return x > y;
10          }
11      }
12      int main(){
13          std::vector<int> v = {1, 2 ,3 ,4, ,5 ,6}
14          std::sort(v.begin, e.end, comp()); //default constructor
15          for (int x : v){
16              std::cout << x;
17          }
18          std::set<int, comp> s; //type, not object
19          for (int x : s){
20              std::cout << x;
21          }
22      }
```

5 Наследование - второй принцип ООП

Некоторые типы могут быть “подтипами” других типов. Производные типы содержат все поля и методы родителей, а также и некоторые свои.

Заметка: Как при описании отношений двух сущностей определить, когда уместно наследование, а когда — композиция? Можно воспользоваться популярной шпаргалкой: сущность А является сущностью Б? Если да, то скорее всего, тут подойдет наследование. Если же сущность А является частью сущности Б, то наш выбор — композиция.

5.1 Public, private, protected inheritance

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

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.

- Public-наследование - означает, что поля, которые достались наследнику от родителей являются public, и к ним можно получить доступ извне.

- 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.

- Protected-наследование - означает, что к приватным полям класса имеют доступ еще и наследники и их друзья. Поля класса могут быть protected.

  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.

5.2 Видимость и доступность (visibility)

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.

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

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

struct Son : Mom {
    void f(double y){};
};

int main(){
    Mom m;
    m.f(); // output: "Mom"
    Son s;
    //s.f(); //CE, this func is inaccesable
    //m.Granny::f(); // CE, as Granny inaccessible
    s.Mom::f(0);
}
```

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;

}
```

5.3 Множественное наследование

```
class Computer {
    public:
        void turn_on() {}
};

class Monitor {
    public:
        void show_image() {}
};

class Laptop: public Computer, public Monitor {};
```

В примере ниже вызов s.a() вызовет CE, так как наследуется метод s.a() от обоих родителей и неясно какой метод должен быть вызван. Возможно приходящее на ум решение сделать в одном из родителей одно из таких полей приватным тоже не работает: несмотря на то, что приватные данные не наследуются, разрешить неоднозначное наследование изменением уровня доступа к данным на приватный невозможно. При компиляции, сначала происходит поиск метода или переменной, а уже после — проверка уровня доступа к ним.

```
class Mother {
public:
    int a = 1;
```

```
4     };
5     class Father {
6     public:
7         int a = 2;
8     };
9     class Son: public Mother, public Father {
10        int s = 3;
11    };
12    int main() {
13        Son s;
14        cout << s.a;
15        //CE: request for member "a" is ambiguous
16    }
```

5.4 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]

```
1     struct Granny {
2         int g = 0;
3     };
4     struct Mother: public Granny {
5         int m = 1;
6     };
7     struct Father: public Granny {
8         int f = 1;
9     };
10    struct Son: public Mother, public Father {
11        int s = 3;
12    };
```

6 Полиморфизм

6.1 Виртуальное наследование

Проблема ромбовидного наследования была решена введением нового типа наследования, виртуального. При таком типе создаётся одна копия родителя. Виртуальное наследование даёт увеличение размера объекта. Pointer ведёт на vtable

```
1     struct Granny {
2         int g = 0;
3     };
4     struct Mother: public virtual Granny {
5         int m = 1;
6     };
7     struct Father: public virtual Granny {
8         int f = 1;
9     };
10    struct Son: public Mother, public Father {
11        int s = 3;
12    };
```

до virtual: [g][m][g][f][s]
после, объект не цельный по памяти: [m_ptr][m][f_ptr][f][s][g]
Минусы:

1. Непривычная работа со static_cast
2. Непривычное расположение в памяти

6.2 Идея полиморфизма

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. пример: операция плюс может быть для чисел, матриц, векторов.

```
1     struct Base {
2         void f() { cout << 1; }
3     };
4     struct Derived: public Base {
5         void f() { cout << 2; }
6     };
7
8     int main () {
9         Base b;
10        b.f(); //1
11        Derived d;
12        d.f(); //2
13        Base bb = d;
14        bb.f(); //1
15        Base& bbb = d;
16        bbb.f(); //(*), 1 is printed - we take parent's version
```

```
17      }
```

6.3 Виртуальные функции

Чтобы в (*) вывелось 2, то достаточно сделать функцию f из Derived virtual: vurtual f() {...}, то есть

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

```
1     struct Base {
2         virtual void f() { cout << 1; }
3     };
4     struct Derived: public Base {
5         void f() { cout << 2; }
6     }
```

Derived автоматически виртуальной

Virtual - основной механизм реализации полиморфизма в C++. Все ООП поддерживают полиморфизм. В Java все функции виртуальные.

Полиморфный тип - тот, в котором определён хотя бы один виртуальный метод.

6.4 Выбор версии между виртуальной и невиртуальной

```
1     struct Base {
2         virtual void f() { cout << 1; }
3     };
4     struct Derived: public Base {
5         void f() const { cout << 2; }//(1)
6         virtual void f() const { cout << 2; } // (2)
7     }
```

(1) - не виртуальный! потому что не полностью совпадает по сигнатуре (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

6.4.1 Ключевое слово override

Виртуальная функция дочернего класса является переопределением, только если совпадают её сигнатура и тип возврата с сигнатурой и типом возврата виртуальной функции родительского класса. А это, в свою очередь, может привести к проблемам, когда функция, которая должна быть переопределением, на самом деле, им не является.

Для решения такого типа проблем и добавили модификатор override в C++11. Модификатор override может использоваться с любым методом, который должен быть переопределением. Достаточно просто указать override в том месте, где обычно указывается const (после скобок с параметрами). Если метод не переопределяет виртуальную функцию родительского класса, то компилятор выдаст ошибку.

```
class A {
public:
    virtual const char* getName1(int x) { return "A"; }
    virtual const char* getName2(int x) { return "A"; }
    virtual const char* getName3(int x) { return "A"; }
};

class B : public A {
public:
    virtual const char* getName1(short int x) override {
        return "B";
    } //(1), CE
    virtual const char* getName2(int x) const override {
        return "B";
    } // (2), CE
    virtual const char* getName3(int x) override {
        return "B";
    } // (3), OK
};
```

(1) - ошибка компиляции, метод не является переопределением (2) - ошибка компиляции, метод не является переопределением (3) - всё хорошо, метод является переопределением A::getName3(int)

6.4.2 Ключевое слово final

Могут быть случаи, когда вы не хотите, чтобы кто-то мог переопределить виртуальную функцию или наследовать определенный класс. Модификатор final используется именно для этого. Если пользователь пытается переопределить метод или наследовать класс с модификатором final, то компилятор выдаст ошибку.

Указывается final в том же месте, в котором и модификатор override, например:

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

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

struct C : public B {
    // CE, redifinition of  B::getName(),that is final
    virtual const char * getName() { return "C";
    }
};
```

В этом коде метод B::getName() переопределяет метод A::getName(). Но B::getName() имеет модификатор final, это означает, что любые дальнейшие переопределения этого метода будут вызывать ошибку компиляции. И действительно, C::getName() уже не может переопределить B::getName() — компилятор выдаст ошибку.

В случае, если мы хотим запретить наследование определенного класса, то модификатор final указывается после имени класса.

6.5 Абстрактные классы и pure virtual классы

До этого момента мы записывали определения всех наших виртуальных функций. Однако C++ позволяет создавать особый вид виртуальных функций, так называемых чистых виртуальных функций (или «абстрактных функций»), которые вообще не имеют определения. Переопределяют их дочерние классы.

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.

```
1    class Parent {
2    public:
3        const char* sayHi() { return "Hi"; } // non-virtual func
4        virtual const char* getName() { return "Parent"; } // virtual func
5        virtual int getValue() = 0; //pure-virtual func
```

Во-вторых, все дочерние классы абстрактного родительского класса должны переопределять все чистые виртуальные функции, в противном случае — они также будут считаться абстрактными классами.

Чисто виртуальные методы можно определять (дисклеймер: не призыв к действию).

6.6 Проблема виртуального деструктора

```
1    struct Base {
2        int* a = new int ();
3        ~Base()  { delete a; cout << "Base" << endl; };
4    };
5
6    struct Derived: public Base {
7        int* b = new int();
8        ~Derived() { delete b; cout << "Derived" << endl; };
9    };
10
11   int main() {
12       Base* b = new Derived();
13       delete b;
14   }
```

Output: Base + получаем утечку памяти

Придётся объявлять деструктор виртуальным.**Если класс сам по себе абстрактный, это никак не означает, что его деструктор (чисто) виртуальный**: virtual GrandBase() = 0.

```
1    struct GrandBase {
2        virtual void f() = 0;
3        virtual ~GrandBase() = 0;
4    };
```

```
5
6       GrandBase::~GrandBase() = default;
7
8       struct Base: public GrandBase {
9           int* a = new int ();
10          ~Base()  { cout << "Base" << endl; };
11      };
12
13      struct Derived: public Base {
14          int* b = new int();
15          ~Derived() { cout << "Derived" << endl; };
16      };
```

6.7 RTTI

Что, с точки зрения компилятора, означает полиморфным? Если тип является полиморфным, то компилятор должен каким-либо образом уметь для каждого вызова виртуальной функции от этого типа отгадывать правильную версию. Доказуемо, что в Compile time делать это невозможно. Пример:

```
1       struct Base {
2           virtual ~Base() = default;
3           virtual void f() { cout << 1; }
4       };
5
6       struct Derived: public Base {
7           void f() override { cout << 2; }
8       };
9
10      int main() {
11          int x; cin >> x;
12          Base b; Derived d;
13          Base& bb = x > 0 ? b : d;
14          bb.f();
15      }
```

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

Статический тип - тип выражения, известный компилятору. 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.

6.8 Dynamic cast

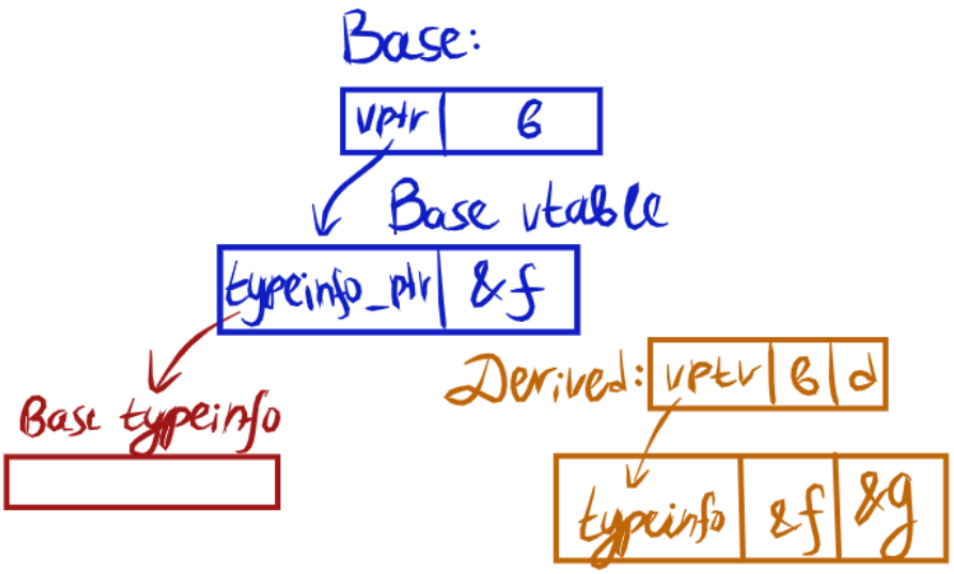
Успешность этого каста зависит от динамического состояния, работает только для полиморфных типов. Примерная схема работы: в runtime проверяем действительно ли тип того, что мы рассматриваем, совместим с типом того, к чему мы хотим привести, если да, то выполняется приведение, иначе RE, а точнее, исключение.

6.9 Виртуальные таблицы и размещение полиморфных объектов в памяти

Каждый полиморфный объект, который расположен в памяти, имеет указатель vptr на кусок статической памяти, которая содержит информацию о типе и создаётся для каждого типа в единичном экземпляре - vtable.

```
1   struct A {}; //sizeof(A) = 1;
2   struct Base {
3       virtual void f() {};
4       int b;
5       void foo() {};
6   };
7   struct Derived: Base {
8       void f() override {};
9       int d;
10      virtual void g() {};
11  };
```

Рассмотрим код и схематично изобразим его конфигурацию в памяти:



Когда вы имеете дело с объектом типа Derived, а потом пишете $Base\&b = d$, делается каст, который создаёт объект, который представляет из себя комбинацию оранжевых vptr и b (из Derived). После, делая b.f() (где b - ссылка на тип Derived, хотя тип у неё - ссылка на Base), мы используем информацию по vptr, чтобы узнать, куда двигаться дальше. О господи как плохо-то всё, лучше просто посмотреть этот кусок, **итог: мы имели дело со ссылкой на Base, но вызовется версия для Derived**.

7 Шаблоны

7.1 Идея

Нужна была возможность писать обобщенные функции - для всех типов сразу. Работает автоматическое определение что такое T. После template <typename T> обязательно идет объявление функции. Тогда область видимости этого template - на эту функцию: объявление и определение.

```
template <typename T>

T max(T a, T b) {
    return a > b ? a : b;
}

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

typename T - тот самый шаблонный параметр.

```
template <typename T>

T max(T a, T b) {
    return a > b ? a : b;
}

int main(){
    std::cout << max(1, 2.0); //CE - conflictive types
    std::cout << max<int>(1, 2.0); // will called with int
}
```

Другое решение проблемы:

```
template <typename T, typename U>

auto max(T a, U b) {
    return a > b ? a : b;
}

int main(){
    std::cout << max(1, 2.0);
}
```

Шаблонные псевдонимы(с c++11):

```
template <typename T>
using mymap = std::map<T,T>

int main(){
    mymap<int> m;
}
```

Шаблонные переменные:

```
template <typename T>
const T pi = 3.14
std:: cout << pi<double>
```

7.2 Шаблонные классы

Шаблонами можно быть не только функции но и классы. Пример шаблонного класса - вектор. Template <typename T> распространяется на все определение класса. Классы нельзя перегружать. Рассмотрим код ниже: метод push_back - это шаблон внутри шаблон, а не шаблон с двумя параметрами

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

//what if we want to definite method outside the class?
template <typename T>
template <typename U>
void vector<T>::push_back(const U& value);
//template <typename T, typename U> is a different thing
```

Замечание: template <typename T> == template <class T>. Здесь class - не то же самое, что подразумевается при объявлении класса

7.3 Перегрузка шаблонных функций

Как работает выбор верссии функции когда есть шаблонные функции. Мудрые люди говорили: "Между частным и общим выбирай частное"- поэтому если у компилятора есть выбор - между отлично подходящей нешаблонной функцией и, собственно, шаблонной, он выберет нешаблонную.

Иллюстрация:

```
template<typename T, typename U>
void f (T x);

void f(int x); // this one will be called

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

2ая эвристика шаблонных функций: "Если есть возможность получить точное соответствие, то это лучше приведения типов".

Неформально: "Если есть две версии, которые одинаково хорошо подходят и одна более частная чем другая, то предпочти ее"

Иллюстрация:

```
template<typename T, typename U>
void f (T x, U y);
```

```
3
4       template <typename T>
5       void f(T x, T y);
6
7       void f(int x, double y);
8
9       int main() {
10          f(0,0);
11      }
```

Комментарий: Вторая функция более частная, и для нее не придется делать неявное приведение типов как в третьей, поэтому компилятор отдает предпочтение второй.

Если есть две функции, так что компилятор не сможет между ними выбрать - одинаковые приоритеты по правилам - то выдастся CE - ambiguous call Пример: f(int, T), f(T, int) и вызов f(int, int).

```
1       template<typename T>
2       void f (T x);
3
4       template <typename T>
5       void f(T& x)
6
7       int main() {
8           int x = 9;
9           int& y = x;
10          f(y); // CE, because f(y) == f(x) as y is a reference and there is
                no difference between reference-argument and non-reference
                argument
11      }
```

Когда компилятору нужно решить по значению ли надо принимать или по константной-/неконстатной ссылке - он не может этого сделать, но когда ему надо решить - по констатной или неконстатной ссылке надо принять аргумент, то в одном случае нужен каст, а в другом нет, и работает 2ое правило перегрузки.

7.4 Специализация шаблонов

Пусть мы написали шаблонный класс, а потом решили, что для какого-то конкретного типа этот шаблон не должен работать, потому что хотим сделать другую реализацию.

Объявим специализацию шаблона в случае, когда T == int

```
1       template<typename T>
2       class s{};
3
4       //full specialization
5       template <>
6       class s<int>{};
7
8       //partial specialization
9       template<typename T>
10      class s<T*>{}; //for all types T that are pointers
```

Специализация "присоединяется"к тому, что написано выше и ближе всего. Частная специализация используется для специализаций указателей, константный/неконстантных ссылок, массивов (<T[]>)

Частичная специализация неприменима для функций - вместо этого есть перегрузка. Пусть у нас есть перегруженные шаблонные функции и. их специализации - сначаал выбирается наилучшая перегрузка, а дальше среди выбранной перегрузки выбирается специализация или сама шаблонная функция.

7.5 Параметры по умолчанию

В шаблонных функциях и классах есть поддержка аргументов по умолчанию.

```
template<typename T = int>

//CE
template<typename T, typename U = int>
void f(T x, U y = 10);
f<int, std::string>(1) // as there is only one argument, default
   argument will be used, but you can't cast string to int
```

Иллюстрация:

```
template<typename T, typename U>
void f (T x, U y);

template <typename T> // among overloaded functions this one is best
void f(T x, T y);

template<>
void f(int x, int y); //specialization will work better

int main() {
    f(0,0);
}
```

7.6 Параметры шаблонов, не являющиеся типами

Шаблонными параметрами не обязательно должны быть типы. В т. ч шаблонным параметром может быть любой объект целочисленного типа, а также char и bool(кроме перечисления) - числовые параметры. Примечательно, разные числа будут считаться разными типами. Типы, которые можно отдавать в шаблонные аргументы должна быть константами, причем Compile-Time.

Кроме того, могут быть шаблонные параметры в шаблонах

```
template<typename T, template <typename> class Container = std::vector>
class Stack{
    Container<T> C;
};

Stack<int, std::vector> s;// vector is a template

template<typename T, typename Container = std::vector<T>>
```

```
    class Stack{
        Container C;
    };
```

Во втором случае второй шаблонный параметр - тип, в первом - шаблон. Несоответствия: typename Container = std::vector - это шаблон,template <typename> class Container = std::vector<T» - это тип.

```
    template<bool N>
    struct s{
    };
    s<3>S;
    s<5>S1;
    // S1 = S will not cause CE!
    //STL container
    std::array<int, 5> a;
    std::Array<int, 10> aa;
    //aa = a will cause CE!

    int x = 5;
    std::array<int, x> // CE, x is not const int
    const int y = x;
    std::array<int, y> //also CE
```

Массивы одинакового размера из STL можно присваивать друг другу. std::array не создается от объектов без дефолтного конструктора.

8 Исключения

8.1 Базовая идея

Из функции можно выйти двумя способами - обычным и выбросом исключения. Выброс делается с помощью команды **throw**. Это оператор, а не управлящее слово. После этой команды генерируется ошибка, и она "бросается". Можно "бросать"любые объекты.

```
1     try{
2         // code that possibly generates a mistakes
3         //if it did we stop that code and go to the section "catch"
4     } catch (const std::out_of_range& err) {
5         //code that's runned in case int is catched
6     }
```

Чтобы поймать любое брошенное исключение, можно в аргументах catch написать многоточие

```
1     try{
2
3     } catch (...) {
4
5     }
```

try..catch ловит только то что было "брошено"командой **throw**

Если в функции вызвалось throw, то переходим на уровень выше, в функцию которая вызывала данную функцию. Если там не было catch чтобы поймать исключение, то происходит переход на уровень выше, до тех пор пока мы не попадем в main или не встретим catch.

8.2 Разница между исключениями и ошибками RE

Далеко не каждая ошибка является исключением. Почти все ошибки - это UB или SegFault. Однако, есть два типа ошибок, которые генерируют исключения:

dynamic_cast: если dynamic_cast произошел неудачно - то выбрасывается ошибка **std::bad_cast**;

new: если компилятор не смог выделить память, то new в этом случае кидает объект типа **std::bad_alloc**.

В случае ошибки следует "бросить"объект класса **std::exception**. У кажодго типа этих исключений есть метод **what**, который покажет в чем именно ошибка. Кроме того при вызове ошибки можно в конструктор передать сообщение, которое выведется при выкидывании исключения.

```
1     try{
2         throw std::out_of_range('Out of the range');
3         v.at(1000000000) = 1;
4     } catch(std::exception& ex){
5         std::cout << ex.what();
6     }
```

Dynamic cast между различными типами ошибок проводится по общим правилам.

8.3 Правила ловли исключений. Catch

В catch запрещены почти все приведения типов. Т.е. если определили что catch "ловит"int, то он не поймает брошенный char и даже unsigned int. **Исключение 1** - приведение типов между родителем и наследником - поймать наследника по ссылке на родителя или по копии родителя можно. **Исключение 2** - приведение const к non-const.

Приведение типов между родителем и наследником стало допустимым, чтобы можно было в типе ошибки указать тип std::exception, и поймать любую ошибку из этого класса. Можно создать собственного наследника, переопределив какой-либо класс(например std::logic_error), тогда будут ловиться исключения как дефолтным типом так и пользовательским.

Catch не работает с перегрузкой. Если после try написано несколько catch-ей, то компилятор выберет первый подходящий, выполняет его, и остальные catch игнорируются, даже если в вызванном catch тоже есть throw, который выкинет исключение.

8.4 Исключение и копирование

Если хотим бросить какой-то локальный объект функции, он должен быть копируемым. Поймать объект можно по ссылке или копии, но не по указателю.

```
1     void f(){
2         Noisy x;
3         throw x; //as x is a local object it will be copied before throwing
4
5         throw Noisy(); //copy-operator won't be called
6     }
7
8     try{
9         f();
10    } catch(const Noisy& x){}
11    //catch(Noisy x){}
```

Чтобы передать пойманное исключение дальше, для дальнейшей обработки, в catch можно написать throw без параметров - "отпускаем лететь дальше то что прилетело". Если же в catch написать throw ex (ex - имя пойманного объекта), то создастся копия этого объекта и уже она бросится, а старый объект удалится.

8.5 Исключения в конструкторах. Идиома RAII

Если конструктор объекта выкинул исключение, то объект считается не до конца созданным, следовательно от него нельзя вызвать деструктор (иначе возникало бы UB), а значит может произойти утечка ресурсов, которые были выделены конструктором до того, как было выкинуто исключение. Решить данную проблему помогают умные указатели, так как мы точно знаем что при любом выходе из функции - штатном или нештатном - будут вызваны деструкторы всех локальных объектов.

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

8.6 Спецификация исключений. Ключевое слово noexcept

С C++11 есть возможность указать в сигнатуре, кидает ли потенциально функция исключение с помощью ключевого слова **noexcept**. Это обещание, которое не проверяется компилятором. Если функция, отмеченная noexcept выкинет исключение, то программа сразу завершится вызовом terminate

```
1    void f() noexcept {}
```

noexcept имеет два значения - спецификатор, в том числе условный, и оператор. Оператор noexcept принимает выражение и возвращает true если выражение безопасно с точки зрения исключений (не содержит операторов которые приводят к выбросу исключений : throw, new, dynamic_cast и вызов функции, которая сама не помечена как noexcept). Оператор noexcept не вычисляет выражение которое в нем стоит, поэтому noexcept(1 / 0) выдаст true.

```
1    void f() noexcept(noexcept(g(1)){}
```

Методы класса, которые гипотетически могут выдать ошибку, даже без исключений не помечаются noexcept (например [ ]).

Noexcept нужно писать для методов: size, empty и тп.

В Catch могут быть только noexcept инструкции

8.7 Function-try block

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

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

8.8 Исключения в деструкторах

При выполнении кода ниже функция g при удалении локального объекта x выкинет исключение, и поскольку в функции f нет catch, то функция f должна завершиться нештатно и отправить исключение дальше наверх по стеку - следовательно она начинает удалять все свои локальные поля и при удалении объекта s выкидывается еще одно исключение. Программа завершается вызовом terminate, так как в C++ не поддерживается несколько летящих исключений. Исключения в деструкторах - это плохо, так как выброс исключения при обработке другого исключения возможен только если исключение выбрасывает деструктор.

С C++11 все деструкторы по умолчанию считаются noexcept функциями и написание выброса исключения в деструкторе сразу приводит к вызову terminate. (Noexcept в деструкторе можно обойти написав noexcept(false), но мы же не хакеры)

```
1    struct Dangerous {
2        int x = 0;
3        Dangerous (int x): x(x){};
4        ~Dangerous(){
5            if (x == 0) {
6                throw 1;
7            }
8        }
```

```
9       };
10
11      void g(){
12          Dangerous s(0);
13      }
14      void f(){
15          Dangerous s(0);
16          g();
17      }
18
19
20      int main(){
21          try{
22              f();
23          } catch(...) {}
24      }
```

8.9 Гарантии безопасности относительно исключений

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.

9 Итераторы

Итераторы - сущности которые позволяют перечислять элементы некоторой последовательности. То что ведет как итератор и есть итератор. Вести себя как итератор - позволят себя разыменовывать, инкрементировать и сравнивать на равенство.

```
1     std::vector<int> v = {1, 2, 3, 4, 5};
2     for(std::vector<int>::iterator i = v.begin(); i != v.ends; ++i){
3         std::cout << *i << " ";
4     }
```

Итераторы нужны если нужно пройтись по контейнеры с нелинейным порядком объектов, например list, map, set, string.

Заметка Итераторы в векторе ломаются если каким-либо образом изменить вектор.

С C++11 работает вот такой вот синтаксис, с помощью которого можно перебрать все элементы контейнера.

```
1     for (int x : v) {
2         std::cout << x;
3     }
```

9.1 Категории итераторов

Есть три вида итераторов

Forward Iterator: однонаправленные итераторы,могут перемещаться только в одну сторону на 1 позицию, перемещение в обратную сторону занимает продолжительное время. Позволяет инкрементировать себя, при этом операция += 1 не определена. Пример: вспомнить после какой строки в стихотворении идет определенная строка;

Bidirectional Iterator двунаправленные итераторы, могут быстро перемещаться на одну позицию как вперед, так и назад. Позволяет не только инкрементировать, но и декрементировать себя - при этом операция += 1 не определена; Пример: станции до и после Новодачной

Random-access итераторы: могут перемещаться быстро на любую позицию в контейнере. Позволяет прибавлять и вычитать из себя любые числа, вычитать два итератора друг из друга и сравнивать на меньше, больше.

Все эти итераторы являются частным случаями **InputIterator** - это итератор, который позволяет лишь раз пройтись по последовательности. Такой итератор есть например у оператора **std::istream**. Если его скопировать и пройтись 2ой раз, не гарантируется что получим ту же последовательность. Остальные 3 итератора это гарантируют.

Контейнеры из STL, которые обладают:

- только ForwardIterator - forward_list, unordered_map, unordered_set.
- BidirectionalIterator - list, map, set
- Random-Access Iterator - vector, deque

Алгоритмы из STL, реализованные с помощью итераторов :

- InputIterator - find()

- ForwardIterator - binary_search()
- BidirectionalIterator - next_permutation()
- Random-Access Iterator - qsort()

9.2 Функции std::advance и std::distance

Функция **std::advance** перемещает итератор на определенное число позиций вперед, ничего не возвращает.

Функция **std::distance** возвращает расстояние между двумя итераторами. Если второй итератор недостижим от первого, то поведение std::distance не определено.

```
1    std::list<int> l = {1, 2, 3, 4, 5};
2    std::list<int>::iterator it = l.begin();
3    std::advance(it, 3);
4    std::list<int>::iterator it2 = l.end();
5    std::cout << std::distance(it, it2); // 2
6    std::cout << std::distance(it2, it); // UB
```

Функции отрабатывают за разное время в зависимости от того, какие итераторы были переданы. В случае FI и BI функции работают за $O(n)$, в случае RA за $O(1)$.

Реализация функции std::advance, которая проверяет тип итератора, и в зависимости от типа отрабатывает за разные время:

```
1    template <typename Iterator>
2    void my_advance (Iterator& it, int n) {
3        if (std::is_same_v<typename std::iterator_traits<Iterator>::
             iterator_category, std::random_access_iterator_tag>) {
4            it += n;
5        } else {
6            for (int i = 0; i < n; ++i; ++it);
7        }
8    }
```

Примечание: std::is_same_v используется для проверки на равенство типов. Просто написать == было бы нельзя! Впрочем, этот код все равно на скомпилируется, потому что если мы вызовем функцию **my_advance** не от RA, то хотя мы и не зайдем в **if**, компилятору нужно будет скомпилировать эту строку, но операцию += не определена для не RA.

Выход 1: после **if** добавить ключевое слово **constexpr** (с C++17) , которое показывает компилятору, что код внутри условия не нужно компилировать, в случае если условие ложно. Важно, что условие должно быть Compile-Time проверяемым

Выход 2: с помощью перегрузки функций (что является костылём)

```
1    template <typename Iterator, typename ItCategory>
2    void my_advance_helper (Iterator& it, int n, Category) {
3        for (int i = 0; i < n; ++i; ++it);
4    }
5
6    template <typename Iterator>
7    void my_advance_helper (Iterator& it, int n, std::
         random_access_iterator_tag) {
8        it += n;
```

```
    }

    template <typename Iterator>
    void my_advance (Iterator& it, int n) {
        my_advance_helper(it, n, typename std::iterator_traits<Iterator>::
            iterator_category());
    }
```

9.3 Const-итераторы

Итератор, которые не позволяют менять объект под собой. Разыменовывая такой итератор, получаем ссылку на константный объект. Константному итератору можно присвоить обычный, но не наоборот

```
    std::list<int> l = {1, 2, 3, 4, 5};
    std::list<int>::const_iterator it = l.  begin  ();
    const std::list<int> cl = {1, 2, 3, 4, 5};
    std::list<int>::iterator it = cl.begin(); // will be const iterator
```

9.4 Реализация класса Итератор

Соглашение: Оператор -> должна возвращать C-Style Pointer. Компилятор сам навесит дополнительную стрелочку и все будет работать корректно.

Чтобы не копипастить код при реализации константный и неконстатных указателей, можно воспользоваться **std::conditional_t<IsConst, const T*, T*** - будет типом первого шаблонного операнда если условие истинно, и второй если ложно. Определен в заголовочном файле <typetraits>. Условие должно быть вычислимо на этапе комиляции - в данном случае это шаблонный параметр.

```
    template <bool isConst>;
    std::conditional_t<IsConst, const T*, T> ptr;
```

41:31

9.5 Reverse-итератор

Если контейнер поддерживает BD-итераторы, то он должен поддерживать и reverse-iterator. Как он работает: например вместо оператора ++() выполняется –(), вместо += будет -=. RI можно сконструировать от обычного итератора, и он ведет себя как обычный итератор за исключением арифм.операций и сравнений

```
    using reverse_iterator = std::reverse_iterator<iterator>

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

Аналогично обычному итератору, можно определить const reverse-итератор - это будет reverse-итератор от const iterator. Есть структура reverse_iterator в STD в заголовочном файле <iterator>, она делает RI от произвольного итератора, она используется в контейнерах.

Зачем использовать? - Например, если хотим вывести все элементы контейнера в обратном порядке, а у нас есть только FI. std::list<int>const_reversed_iterator

```
for (auto it = v.rbegin(); it != v.rend(); ++it) {
    std::cout << *it;
}
```

rbegin() - возвращает RI на последний элемент, rend() - вернет фиктивный итератор на элемент, перед первым

У RI есть метод **base()**, с помощью которого можно получить исходный итератор

```
Iterator base() const{
    return iter;
}
```

9.6 Output-итераторы

Итератор, писать в который можно с помощью алгоритмов по типу std::copy - итератор гаранитрует, что если по нему таким образом пойти писать, то ничего плохого не случится. В стандартных контейнерах output-итераторы не являются OI, так как не гарантируется корректность (что мы не выйдем за границы контейнера). Как получить OI на контейнер - существует специальный адаптера из СБ, которые позволят писать в контейнеры с помощью итераторов.

```
std::list<int> l = {1,2,3}
std::vector<int> v;
std::copy_if(l.begin(), l.end(),std::back_inserter(v),isEven()};
```

Функция std::back_inserter(v) создает back_insert-итератор от данного объекта . Это класс, шаблонным парметром которого является контейнер. В себе он хранит ссылку на контейнер. Разыменование BI-итератора дает не то что под итератором, а снова сам итератор. Он умеет делать присваивание себе элементов контейнера, который лежит под BI-итератором , это присванивание возвращает ссылку на back_insert_iterator. Получили итератор с нужным свойством - он только добавляет элементы в контейнер. Почему в функции не могли явно прописать итератор? - потому что если у нас контейнер с кучей принимаемых значений, то код становится очень массивным и неудобным для восприятия. Компилятор сам понимает, какой тип контейнера передан в функцию.

```
template <typename Container>
class back_insert_iterator {
    Container& c;
public:
    back_insert_iterator(Container& c): c(c) {}
    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) {
        c.push_back(value);
        return *this;
    }
};
```

```
17
18      template <typename Container>
19      back_insert_iterator<Container> back_inserter(Container& c) {
20          return back_insert_iterator<Container>(c);
21      }
```

Помимо back_insert-итератора есть front_insert-итератор и просто insert-итератор (который принимает контейнер в конструктор и итератор на этот контейнер и вместо push_back делает insert по итератору в этот контейнер). Т.е например для set-а подойдет только insert-итератор, так как в этих контейнерах нельзя делать push_back

9.7 Итераторы над потоками - stream iterators

Концепция итераторов не привязывается к контейнерами, это абстракция, которая позволяет перемещаться по последовательности. Итерируемы являются, например, еще потоки. Хотим из потока читать так, как будто это итератор

```
1       std::vector<int> v;
2       std::istream_iterator<int> it(std::cin);
3       for (int i = 0; i < 5; ++i, ++it) {
4           v.push_back(*it);
5       }
```

Замечание: При таком вводе нужно будет ввести 6 чисел, так как первое считывание происходит во 2ой строке, потому что в конструкторе есть считывание. А последний элемент игнорируется.

В конструкторе должно считать одно значение, чтобы при разыменовании итератора было что возвращать. istream-итератор - пример input-итератора, который не является ForwardIterator

Так же есть ostream-итератор для вывода в поток.

```
1       std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ")
            );
```

Реализация:

```
1       template <typename T>
2       class istream_iterator {
3           std::istream& in;
4           T value;
5       public:
6           istream_iterator(std::istream& in): in(in) {
7               in >> value;
8           }
9           istream_iterator<T>& operator++() {
10              in >> value;
11          }
12          T& operator*(){
13              return value;
14          }
15      };
16      // children:
17      // std::ifstream in("input.txt");
18      // std::istringstream iss(s);
```

10 Стандартные контейнеры

Общие правила: При реалзиации контейнеров должны даваться строгие гарантии безопасности относительно исключений.

10.1 Обзор контейнеров

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

Container	indexating[]	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	$O(1)$ m	-	$O(1)$ m	$O(1)$ m	$O(1)$ m	FI

Примечание: $O(1)$ m - 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.
Заметим, что у **forward_list, list** и **deque** есть метод push_front, который работает за $O(1)$.

10.2 std::vector

```
1   template <typename T, typename Alloc = std::allocator<T>>
2   class Vector{
3       T* arr;
4       size_t sz;
5       size_t capacity;
6       Alloc alloc;
7
8       using Alloctraits = std::alloctator_traits<Alloc>;
9   public:
10      Vector(size_t n, const T& value = T(), const Alloc& alloc = Alloc())
            ;
11
12      T& operator[](size_t i) {
13          return arr[i];
14      }
15      T& at(size_t i) {
16          if (i >= sz) throw std::out_of_range("...");
17          return arr[i];
18      } //also this methods but for const Vector
19
20      size_t size() const {
21          return sz;
22      }
23      size_t capacity() const {
24          return cap;
25      }
26
27      void resize(size_t n, const T& value = T());
```

```
28          void reserve(size_t n);
29      };
```

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 воспользоваться методом **std::shrink_to** из <**vector**>

```
1       void reserve(size_t n) {
2           if (n <= cap)
3               return;
4           T* newarr = new T[n];
5           for (size_t i = 0; i < sz; ++i) {
6               newarr[i] = arr[i];
7           }
8           delete[] arr;
9           arr = newarr;
10      }
```

В коде 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 (конструктора по умолчанию типа T может просто не быть).

```
1       void reserve(size_t n) {
2           if (n <= cap)
3               return;
4           //T* newarr = new T[n]
5           T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
6           for (size_t i = 0; i < sz; ++i) {
7               //newarr[i] = arr[i]
8               new(newarr + i) T(arr[i]);
9           }
10          delete[] arr;
11          arr = newarr;
12      }
```

Реализация на хорб уже выделяет необходимое количество байт для хранения, но теперь у нас будет SegFault. Мы не можем делать присваивание к newarr[i], так как в реальности под newarr[i] лежит сырая память. Таким образом, нам нужно вызвать конструктор T по данному адресу от данного объекта. Для этого существует специальный синтаксис: placement-new. Но проблема не устранена, так как delete[] тоже будет SegFault, так как в arr в реальности не лежит cap объектов, а sz.

- Реализация с помощью *uninitialized_copy*

```
1       void reserve(size_t n) {
2           if (n <= cap) return;
3           T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
4           try {
5               std::uninitialized_copy(arr, arr + sz, newarr);
6               // does not work with allocator
```

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

        ////// 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);
        } //////
    }
```

- Самая хорошая реализация с Allocator & std::move

```
    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;
```

```
26      }
```

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.

```
1     void push_back(const T& value) {
2         if (sz == cap)
3             reserve(2 * cap);
4         //new(arr + sz) T(value);
5         AllocTraits::construct(alloc, arr + sz, value);
6         ++sz;
7     }
8     void pop_back(const T& value) {
9         //(arr + sz - 1)->~T();
10        AllocTraits::destroy(alloc, arr + sz - 1);
11        --sz;
12    }
13    void resize(size_t n, const T& value = T()) {
14        if (n < cap) reserve(cap);
15        /.../
16    }
```

10.3 std::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.

```
1     templtate <typename U>
2     void f(const U&) = delete;
3
4     int main() {
5         vector<bool> vb(10, false);
6         vb[5] = true;
7         f(vb[5]);
8     }
```

В данном случае компилятор начнет ныть, что так нельзя вызывать f от типа, который идален. Так мы заставим компилятор спалиться какой у него тип для vb[5]. Мы увидим, что **U = std::_Bit_reference**. Как же это работает?

```
1     templtate <>
2     class Vector<bool> {
3         int8_t* arr;
4         size_t sz;
5         size_t cap;
6
7         struct BitReference {
8             int8_t* cell;
9             uint8_t num; // pos in this cell
```

```
10
11              BitReference& operator=(bool b) {
12                  if (b) {
13                      *cell |= (1u << num);
14                  } else {
15                      *cell &= ~(1u << num);
16                  }
17                  return *this;
18              }
19
20              operator bool() const {
21                  return *cell & (1u << num);
22              }
23          }
24
25      public:
26          BitReference operator[](size_t i) {
27              return BitReference{arr + i / 8, i % 8};
28          }
29      }
```

Структура BitReference такаия хитрая, что она позволяет, присваиваяя экземпляру себя, менять исходный вектор.

10.4 stack, queue, priority_queue

Это **адаптеры над контейнерами** (иначе - класс обертки над контейнерами), содержат в себе лишь простейшие методы. Стек содержит всего три метода: *push*, *pop*, *top*

Обычно контейнер по умолчанию это дек, так как в нем не инвалидируются итераторы, но вообще может быть любой sequence-контейнер, который поддерживает операции *push_back*, *pop_back*, *back*. У стека два шаблонных параметра: значения, которые принимает контейнер и собственно сам контейнер. Если шаблонный параметр в контейнере не совпадает с шаблонным параметром в самом стеке - возникает UB.

```
1       template<typename T, typename Container=std::deque<T>>
2       private:
3           Container container; // is a type already, no need to point the
              template parametr
4       public:
5           void push(const T& value){
6               container.push_back(value);
7           }
8           void pop;
9           T& top();
10          const T& top() const;
```

Замечание: Почему бы не сделать так, чтобы pop возвращал то значение элемента, которое он собирается удалить? **Ответ:** для улучшения эффективности - чтобы отдать элемент pop приходилось бы копировать элемент (он не может отдать ссылку на этот элемент - он же его вот-вот удалит), а в случае непримитивного типа, копирование может быть довольно долгим. Но значение удаляемого элемента не всегда нужно, поэтому могло бы получаться ненужное копирование элементов.

Очередь поддерживает push и front. Приоритетная очередь при вызове метода front выдает элемент с наименьшим приоритетом, при добавлении нового элемента, происходит просеивание по приоритетам. **Над этими контейнерами нет итераторов.**

10.5 list

Связный список, хранит внутренний тип *Node* для хранения "вершинок"списка. Ноды хранят элемент листа и указатели на предыдущую и следующую вершинку. В полях листа хранится указатель на начало списка.

Хранить список можно так: сделать фейковую Ноду (созданную из сырой памяти - выделить память через reinterpret_cast к *Node* и руками проставить указатели на prev и next) которая будет связывать голову и хвост листа.

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)

forward_list - односвязный список,в котором отсутствует метод *back*, *pop_back*

10.6 map

Это упорядоченный ассоциативный массив, который хранит пару: ключ-значение.

Map - это красно-черное дерево (сбалансированное двоичное дерево поиска). В нем тоже есть структура **Node**, которая хранит ключ-значение как пару, указатель на родителя, и указатели на двух детей. Итератор в map это указатель на **Node**. Под итератором лежит пара ключ-значение. Еще есть компаратор.

Важно! Ключ это константный тип (const key), так как от ключа зависит положение в дереве.

В map есть методы **find** - возвращает итератор, **count** - считает количество ключей для значения (то есть 0 или 1), **insert**, **at**, **lower_bound**, **upper_bound**, **equal_range**. Важно понимать, что ++it работает за $log(n)$, но при этом проход по всему мэпу работает за линию, так как там будет мало больших $log(n)$.

10.7 unordered_map

Тот же мэп, но не гарантируется порядок ключей. Параметры: ключ-значение, хеш, equal_to<key>.

Производительность у него лучше, так как ключ хранится в хеше (методом цепочек). В массиве лежат указатели на односвязные списки. Если в бакете уже что-то лежит, то проверяем нет ли в бакете уже этого ключа (для этого используем $equal\_to < key >$).

Node представляет из себя пару ключ-значение, указатель на следующую *Node* и еще есть $uint32\_t$ *chached* - обозначает хеш текущего ключа.

Итератор не может просто шагать по хеш-таблице (так как элементов может быть мало). На самом деле все эти односвязные списки связаны в один большой. Поэтому здесь нужен *cached* - чтобы понимать в каком мы бакете. Новый бакет направляется в голову глобального листа. Для удаления либо мы должны знать предшественников для каждого бакета, либо все хранить в глобальном двусвязном списке.

На каждое добавление *Node* нужно обращение к опретору new (как в list с *Node*) - это такой компромисс между производительностью и функциональностью.

10.8 Инвалидация итераторов

Допустим у нас есть вектор и свободное пространство рядом с ним. Если cap закончилось, то вектор реаллоцирует свой storage: мы перекладываем все элементы и уничтожает 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.

```
1     vector<int> v;
2     v.push_back(1);
3     vector<int>::iterator it = v.begin();
4     int* p = &v.front();
5     int& r = v.front();
6     v.push_back(2);
```

При этом list не инвалидирует итераторы, ссылки и указатели.
В случае deque портяться только итераторы.

invalidation rules

	iterators	pointers/references
vector	NO	NO
deque	NO	YES
list	YES	YES
map/set	YES	YES
unordered_map/set	YES*	YES

(YES - неинвалидируется и можно использовать)
(YES* - неинвалидируется, если не было rehash)

11 Аллокаторы

11.1 Перегрузка new/delete

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 отвечают выделению памяти**. Конструкторы будут неизбежно вызваны после выделения памяти. 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, которая 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)
17      }
```

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.

Замечание: Оператор new и функция оператор new - разные вещи. Функция - выделяет память, оператор - выделяет память и вызвает конструкторы.

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

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) А еще можно делать операторы с кастомными параметрами - тогда при вызове new с такими параметрами, будет вызываться перегруженный оператор, но стаднартный при этом так же будет работать.

11.2 Placement new

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

```
1       S* p = reinterpret_cast<S*>(operator new(sizeof(S)));
2       S* p1 = reinterpret_cast<S*>(operator new(sizeof(S)));
3       new(p) S(); //defailt construct will be called
4       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.

```
1    void* operator new (size_t, S* p){
2        return p;
3    }
```

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.

```
1    S* ptr = new(mystruct) S();
2    operator delete(ptr, mystruct);
3    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).

Еще есть **nothrow new** - operator new, который не кидает std::bad_alloc, в случае ошибки возвращает nullptr.

```
1    T* p = new (std::nothrow) T(n);
```

11.3 Идея аллокаторов

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

```
1    template<typename  T>
2    struct allocator{
3        T* allocate(size_t n) {
4            return ::operator_new(n * sizeof(T));
5        }
6
7        void deallocate(T* ptr, size_t n) {
8            ::operator delete(ptr);
9        }
10       ////
11       template<typename... Args>
12       void construct(T* ptr, const Args&... args){
13           new(ptr) T(args...);
14       }
15       void destroy(T* ptr){
16           ptr->~T();
17       }
18   }
```

Вообще в deallocate надо вернуть в качестве параметра такое же значение n, которое передавалось в allocate.

11.4 Нестандартные аллокаторы

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 укзатель 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)

11.5 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 определен этот, то выщывается он, иначе вызовется метод, определенный в **allocator_traits**. Структура со стаическими методами - ссылка на аллокатор и то что нужно передать аллокатору (из-за того что там только методы, то нельзя создать объект этого класса)

11.6 Rebinding allocators

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

```
1     template<typename T, typename Alloc = std::allocator<T>>
2     class list{
3         class Node{};
4         std::allocator_traits<Alloc>::rebind_alloc<Node> alloc;
5     public:
6         list(const Alloc& alloc = Alloc()) : alloc(alloc) {};
7     };
```

11.7 Копирование и присваивание аллокаторов друг другу

Что вообще такое копирование аллокатора (что значит инициализировать один аллокатор другим аллокатором)?

Допустим у нас PoolAllocator. Если мы копируем аллокатор, то пуул копировать не надо, так как новый аллокатор должен уметь освобождать то, что выделил старый аллокатор.

```
1     alloc1 == alloc2
2     // this means
3     T* ptr = alloc1.allocate(1);
4     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.

11.8 Поведение аллокатора при копировании и присваивании контейнера

Нужно ли нам в таком случае делать копию аллокатора или необходимо создать новый.

```
1    vector<int, PoolAlloc> v1;
2    // .... //
3    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 пулл.

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 пулл).

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!

```
1    template <typename T, typename Alloc = std::allocator<T>>
2    class Vector {
3        T* arr;
4        size_t sz, cap;
5        Alloc alloc;
6
7        using AllocTraits = std::allocator_traits<Alloc>;
8    public:
9        Vector(size_t n, const T& val = T(), const Alloc& alloc = Alloc());
10
11       Vector<T, Alloc>& operator=(const Vector<T, Alloc>& other) {
12           if (this == &other) return *this;
13
14           for (size_t i = 0; i < sz; ++i) {
15               pop.back();
16               //AllocTraits::destroy(alloc, arr + i);
17           }
18           AllocTraits::deallocate(arr, cap);
19
20           //main decision
21           if (AllocTraits::propagate_on_container_copy_assignment::value
                 && alloc != other.alloc) {
22               alloc = other.alloc;
23               // what if the exception appear here??
```

```
24              }
25
26              sz = other.cap;
27              cap = other.cap;
28
29              AllocTraits::allocare(alloc, other.cap);
30              for (size_t i = 0; i < sz; ++i) {
31                  push_back(other[i]);
32              }
33               return *this;
34          }
35      }
```

Отныне мы все чаще и чаще будем обращаться к разделу **named requirements** на cpp.reference

Например, наша реализация list должна быть написана согласно *AllocatorAwareContainer*.

12 Move-семантика

12.1 Мотивация, проблемы которые привели к ее изобретению

Придумано в C++11. Рассмотрим функцию **push_back в векторе**

```
1     void push_back(const T& value){
2         if(sz == cap) reserve(2 * cap);
3         AllocTraits::construct(alloc, arr + sz, value);
4         ++sz;
5     }
```

Проблемы:

- Рано или поздно у нас при construct вызывается placement new - $new(arr+sz)T(value)$ - т.е. конструктор копирования и, таким образом, у нас создается два объекта (первый - временный, который мы передаем в push_back и второй, который мы кладем в вектор).
- 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)
- При методе swap с большими объектами (контейнерами) у нас вообще делается копирование 3 раза (тройное пересоздание).
- Пусть есть функция, результатом которой является новый объект типа T.
  - При вызове этой функции в другой части кода $MyHeavyTypeobject = createObject()$; - будет все ок, так как произойдет Copy Elision (оптимизация компилятора).
  - 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.

Решение проблемы с push_back в векторе 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.

```
1     template<typename... Args>
2     void emplace_back(const Args&... args){
3         if(sz == cp) reserve(2 * cp);
4         AllocTraits::construct(alloc, arr + sz, args...);
5     }
```

Как отличаются вызовы функций:

```
1     std::vector<std::string> v;
2     v.push_back(std::string('3', 'a');
3     v.emplace_back('3', 'a');
```

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

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 константной ссылке, будут вызваны лишние копирования.

12.2 Функция std::move и ее применение

Рассмортим функцию swap. С применением функции std::swap все действия происходят за $O(1)$ для всех библиотечных типов. При таком конструировании(а это именно оно!) не будет просходить копирования. Что будет происходить: у объекта x заберется все содержимое, он станет пустым и за $O(1)$ все поля переместятся на y.

```
1    template<typename... Args>
2    void swap (T&x, T& y){
3       T temp = std::move(x);
4       x = std::move(y);
5       y = std::move(tmp);
6    }
```

Заблуждения относительно функции move:

1. Строка std::move(s), результат которой никуда не присваивается, никак не изменит объект s.

2. Обращаться к объекту, от которого была вызвана функция после собственно вызова можно, это не будет считаться за UB. После вызова объект будет в валидном состоянии, просто пустым.

Итак, пока что можно использовать std::move в случае, если у нас есть не временный, именованный объект A, и мы хотим вызвать от него конструктор или присвоить его другому объекту, забрав содержимое объекта A за $O(1)$.

12.3 Move-семантика в классах, правило пяти

Если в пользовательском классе не определить move-семантику, то вызов функции std::move будет работать столько же, сколько операция копирования. Для компилятора std::move - это сигнал, что при определенном контексте кода и если тип объекта поддерживает move-семантику, он уполномочен забрать информацию из объекта, от которого вызван std::move, обнулив объект.

При несоблюдении требований у компилятора появятся такие вопросы:

Я могу обнулить поля объекта, но зачем?

Я мог бы обнулить поля объекта, но объект не поддерживает такое поведение.

Вызывать функцию std::move имеет смысл только от классовых объектов, так как move-семантика поддерживается только классами, следовательно вызывать функцию от переменной типа int бессмысленно, в результате переменная не занулится

Move-constructor: Принимает на вход rvalue reference

```
1    class String{
2        char* str = nullptr;
3        size_t sz = 0;
```

```
4         void swap(String& s) {
5             std::swap(sz, s.sz);
6             std::swap(str, s.str);
7         }
8     }
9
10    String(String&& s) : str(s.str), sz(s.sz){
11        s.sz = 0;
12        s.str = nullptr;
13    }
```

Move-assignment operator: Принимает на вход rvalue reference

```
1     String& operator=(String&& s) {
2         String new_copy = std::move(s); //move-constructor
3         swap(new_copy);
4         return *this;
5     }
```

Правило пяти: Если в классе реализован нетривиальный деструктор и/или нетривиальный конструктор копирования, и/или оператор копирующего присваивания и/или конструктор перемещения и/или перемещающий оператор присивания, желательно определить явно все пять.

Как и с операторами копирования, компилятор умеет автоматически генерировать оператор и конструктор перемещения с помощью **default**. Если этого не сделать, но при этом реализовать какой-то из операторов копирующего присваивания или присвающего перемещения, то "противоположный"компилятор уже не будет автоматически генерировать (будет считать его нетривиальными). Аналогично и для конструкторов.

Однако иногда даже с помощью default может быть сгенерирован неправильный конструктор перемещения, как например конструктор перемещения для нашего класса String

Как происходит генерация по умолчанию: от каждого поля обхекта вызывается std::move. Если полями объектов являются только другие объекты, у которых есть move-семантика или хотя бы конструктор копирования - то все хорошо, если числа - то у объекта, от которого мы конструировались, числоые переменные не обнулятся, но в целом это может быть не так страшно. Однако, если среди полей есть c-style указатели, то конструктор сработает совсем неправильно, указатель не обнулится, и будет проблема двойных указателей (с shared_ptr все сработает правильно, так как это объект класса с реализованной move-семантикой)

Неявно сгенерированный конструктор (implicitly-declared move constructor) может быть сразу помечен как **deleted**, если хотя бы один объект-поле класса не может быть перемещен(т.е конструктор пермещения для этого класса явно запрещен **deleted**, недоступен(приватный) или двусмысленнен), а так же если у класса есть родитель, который не может быть пермещен (вспомним, что при наследовании конструкторы вызываются иерархически) или у родителя есть deleted деструктор.

Отступление: вернемся к push_back вектора. Перегрузим функцию для вызовов от std::move.

```
1     void push_back(T&& value) {
2             if (sz == cap) reserve(2 * cap);
3             AllocTraits::construct(alloc, arr + sz, std::move(value));
4             ++sz;
5     }
```

Value будем отдавать в конструктор не копированием а перемещением.

С emplace_back так проблему решить нельзя, поскольку там в функцию передаются несколько аргументов и мы не знаем заранее какие из них можно пермещать, а какие нет. Применить ко всем std::move нельзя, так как пользователь возможно хочет дальше продолжать работать с объектами-аргументами, а мы их можем испортит.

Вернёмся к реализации метода reserve в векторе и используем там move-семантику.

12.4 Понятия lvalue и rvalue

Эти понятия применимы не к объектам и не к типам объектов, а к expressions.

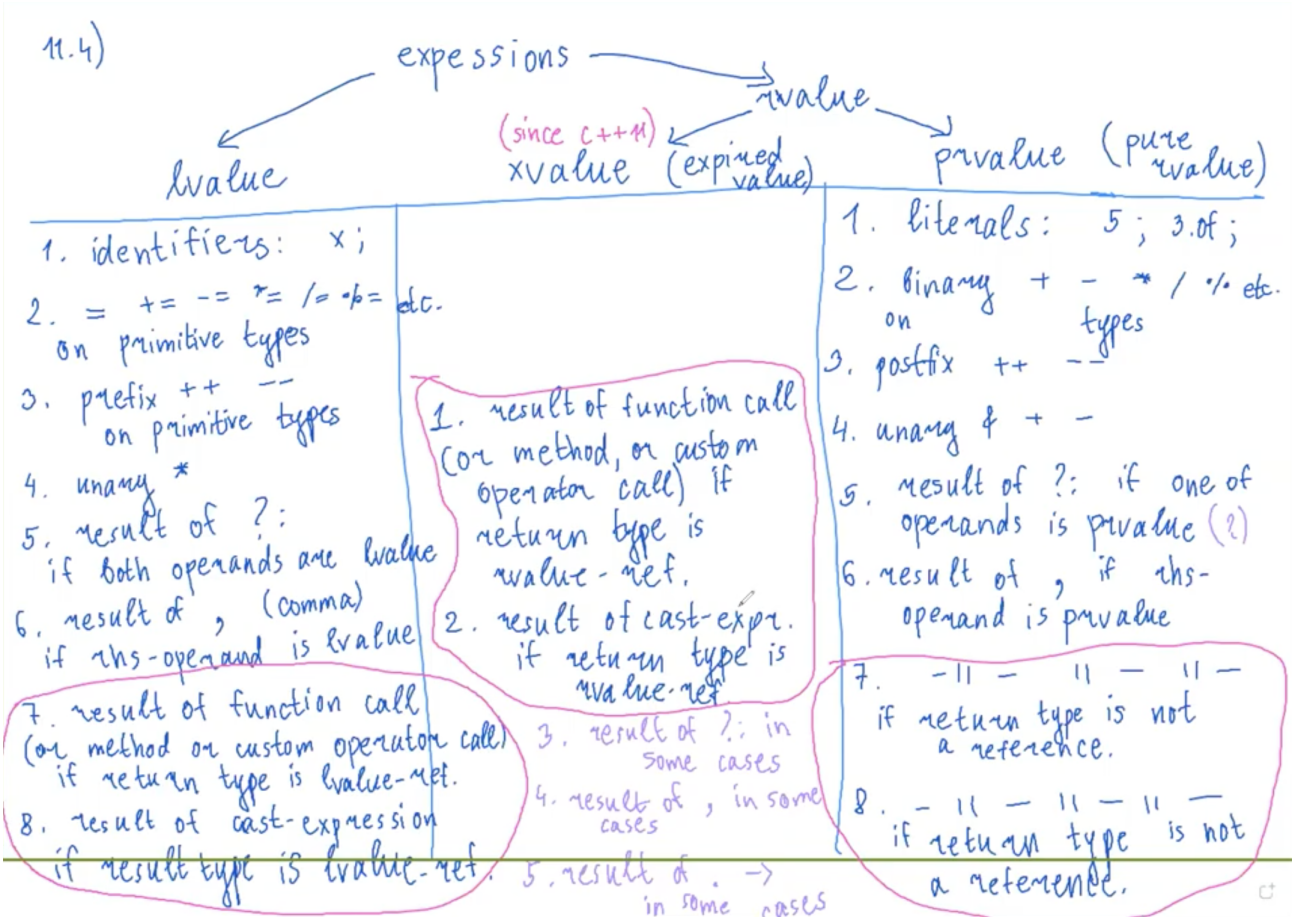
Любой объект может быть как lvalue так и rvalue в зависимости от контекста, хотя опять-таки нельзя говорить про rvalue и lvalue относительно объектов. Интуитивно rvalue - выражение, которе представляет из себя создание нового объекта (то, что не может стоять слева от знака равенства). А lvalue - выражение,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

Например, константные объекты это lvalue, но им присваивать нельзя, или просто объекты, у которых не определён оператор присваивания. Rvalue - только что созданный временный объект $Object()$;. ***Противоположный пример,*** когда мы делаем обращение к $vector < bool >$ с помощью [], мы получаем не ссылку, а временный объект BitReference, к которомы мы и должны присваивать, чтобы изменить значение.

Важно: Rvalue ссылка далеко не всегда rvalue, как и lvalue ссылка не обязательно lvalue, то есть $lvalue\&rvalue$ это виды выражений (синтаксическая конструкция, составленная из переменных, литералов и операторов), а не объектов или типов.

Замечание: Теперь у нас есть два вида ссылок:

- $type\&\&$ - Rvalue reference
- $type\&$ - Lvalue reference



12.5 rvalue references

Неконстантная lvalue reference позволяет себя инициализировать только значениями lvalue Rvalue references позволяют себя инициализировать только rvalue-выражениями. Как и в случае констаной lvalue ссылки, в случае (1) и (2) произйодет продление жизни объекта, переменная создастя и будет дить на стеке пока идет область видимости переменной. В случае (3) присвоили rvalue reference значение, которое лежало в x, это уже не инициализация. (4) - тоже окей, по первому пункту определения,так как rrx - идентификатор, то есть lvalue (и при этом rvalue ссылка одновременно).

```
1     int main() {
2         int x = 0;
3         int& rx = x;
4         // int& rx = 1; WRONG!
5         const int& crx = 1; //(1)
6         int&& rrx = 1; // (2)
7         // int&& rrx = x; WRONG!
8         rrx = x; //(3)
9         rrx = 5;
10        x = 1; //rrx is still 5;
11
12        int& another_ref = rrx;//(4)
13        int&& another_ref_2 = rrx; // WRONG!
14    }
```

В остальном, rvalue ссылки ведут себя как обычно, то есть если их проинициализиорвоали объектом, то они поменяются, когда объект поменяется

```
1    int main() {
2        int&& rrx = 5;
3        const int&& crrx = rrx;
4
5        const int&& const_wrong_rrx = 1;
6        int&& wrong_rrx = const_wrong_rrx;//WRONG, discard qualifiers
7    }
```

Что если нам захочется перепривязать объект от lvalue ссылки к rvalue? Использовать std::move - явное указание, что можно "убить"этот объект (просто так lvalue-объект компилятор в rvalue-функцию не отдаст, ну или трактовать как rvalue объект

```
1    int main() {
2        int x = 0;
3        int& rx = x;
4        int&& rrx = std::move(rx);
5    }
```

Замечание: если есть конструктор от const lvalue-ссылки и от rvalue-ссылки, компилятор отдаст rvalue объект во вторую функцию - сработает перегрузка, и хотя обе функции вроде бы подходят, второй случай считается perfect match

12.6 Universal references

Некоторые функции должны уметь принимать каечтсве параметров как lvalue-reference, так и rvalue-reference Синтаксис должен быть в точности таким, T должно быть шаблонным параметром функции.

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

Сам x здесь - это lvalue, но в зависимости от того, от чего будет вызвана эта функция, тип x может стать как выражением типа lvalue так и rvalue.

Так, в f(5) 5 - rvalue, тип **T** будет равен int, **decltype(x)** = int

int y = 5; f(y); тип **T** будет равен int, тип x по идее должен быть тип T + &&, однако на самом деле работают **reference collapsing rules** и **decltype(x)** = int;

Правило: Если в universal reference отдали lvalue-ref, то тип T становится не типом параметра, а типом параметра + &.

Reference collapsing:

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

Универсальные ссылки считаются более хорошим соответствием чем любые другие ссылки - правило частное лушче общего не работает

12.7 std::move в контексте rvalue и lvalue ссылок

Реализация

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

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

12.8 Perfect forwarding problem и std::forward

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 RR и скорпировать остальные (к ним нельзя применять std::move, так как их передали в f не как RR, 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**

12.9 Исключения

Когда мы заменили в методе reserve для вектора копирование на move-семантику, мы перестали гарантировать безопасность относительно исключений. Перекладывать элементы обратно нельзя. Используется функция **std::move_if_noexcept**, которая определяет, является ли move-конструктор noexcept или нет. Если не помечен как noexcept, то вектор не будет делать move. Если у вектора не будет конструктора копирования, то тогда он будет делать move и потеряется безопасность относительно исключений(даже если по факту конструктор noexcept).

Реализация move_if_noexcept

```
template<typename T>
auto move_if_noexcept(T& x) -> std::conditional<std::
    is_no_throw_move_constructible_v<T>, T&&, const T&> {
```

```
3         return std::move(x);
4     }
```

В случае истины вернется rvalue, иначе lvalue, в зависимости от того, является ли move constructor безопасным. В 16.6 рассказывается подробнее об этих функциях

12.10 Reference qualifiers

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.

```
1     struct S{
2         void f() &{...}
3         void f() &&{...}
4     }
```

Если стоит const & в качестве квалификатора и не определен для &&, то можно будет вызывать от rvalue, но для простого & так нельзя будет сделать, как вызывать && от lvalue.

13 Вывод типов - type deduction

13.1 auto

Проблема: слишком длинные названия типов, например в случае std::unordered_map с нестандартным аллокатором, или например, в том же типе можно случайно забывать квалификатор const у ключа, и как следствие можно получить лишние копирования например в range-based loop. Решение этих проблем- ключевое слово **auto**, которое сообщает компилятору, что тип переменной должен быть установлен исходя из типа инициализируемого 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.

13.2 decltype

В компайл-тайм возвращает тип выражения. Хорош тем, что не отбрасывает , , *, const/volatile, 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.

```
1    int x = 7;
2    decltype(x++) u = x;
```

Если внутри decltype стоит выражение типа prvalue типа T, то тип declype(expression) - T

Если внутри decltype стоит выражение типа xvalue типа T, то тип declype(expression) - T&&

Если внутри decltype стоит выражение типа lvalue типа T, то тип declype(expression) - T&

13.3 type deduction for return type

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

```
1    auto f(int& x){
2        if (x > 5) return x;
3        else return 0.0;
4    } // 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. Auto отбросит ссылки, а auto наоборот навесит лишнюю. Написать auto(f(x)) тоже нельзя, так как на данном этапе x еще не определен. Решение проблемы: trailing return type

```
1    auto g(int& x) -> decltype(f(x)){
2        return f(x);
```

```
3     }
```

С C++14 возможен вот такой синакисис, который говорит компилятору: выведи тип самостоятельно, но не по правилам auto, а по правилам decltype

```
1     template <typename Container>
2     decltype(auto) g(const Container& cont, size_t index)){
3         std::cout << "...";
4         return cont[index];
5     }
```

13.4 Structured binding

С C++17 существует синтаксис который позволяет получать ссылки на пары, копии пар, кортежей и т.д

```
1     std::pair<int, std::string> p(5, "abc");
2     auto [a,b] = p;
3     auto& [c,d] = p;
4     std::cout << a << b;
```

14 Умные указатели

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).

Вот в каких случаях нужно использовать тот или иной указатель:

- **unique_ptr**: должен использоваться, когда ресурс памяти не должен был разделяемым (у этого указателя нет конструктора копирования), но он может быть передан другому unique_ptr
- **shared_ptr**: должен использоваться, когда ресурс памяти должен быть разделяемым (имеется в виду, что когда на одну и ту же область памяти может указывать несколько указателей именно shared_ptr типа).
- **weak_ptr**: содержит ссылку на объект, которым управляет shared_ptr, но не осуществляет подсчет ссылок; позволяет избавиться от циклической зависимости

Все находится в библиотеке <memory>

14.1 std::unique_ptr

std::unique_ptr нельзя копировать, соответствующий конструктор и оператор у них удалены, но зато можно мувать (поэтому вектор из UP корректен). Присвоить значение этому указателю можно только в момент объявления (т.е. инициализация умного указателя должна быть в момент объявления).

Примерная реализация

```
1   template<typename T, typename Deleter = std::default_delete<T>>
2   class unique_ptr {
3   private:
4       T* pointer;
5   public:
6       unique_ptr() {
7           pointer = nullptr;
8       }
9
10      explicit unique_ptr(T* ptr): pointer(ptr) {}
11
12      unique_ptr(const unique_ptr<T>& other) = delete;
13
14      unique_ptr<T>& operator=(const unique_ptr<T>& other) = delete;
15
16      unique_ptr(unique_ptr<T>&& other)  noexcept {
17          pointer = std::move(other.pointer);
18          other.pointer = nullptr;
19      }
20
21      unique_ptr& operator=(unique_ptr<T>&& other)  noexcept {
```

```
22              delete pointer;
23              pointer = std::move(other.pointer);
24              other.pointer = nullptr;
25          }
26
27          ~unique_ptr() {
28              delete pointer;
29          }
30
31          T& operator*() const {
32              return *pointer;
33          }
34
35      };
```

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).

14.2 std::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.

Ниже приведена реализация уже со всеми модификациями и с учетом существования weak_ptr

```
1       template<typename T>
2       class shared_ptr{
3       private:
4           T* ptr;
5           ControlBlock<T>* inner_block = nullptr;
6
7           template<typename  U>
8           friend class weak_ptr;
9
10          template <typename U, typename ...Args>
11          friend shared_ptr<U> make_shared(Args&& ...args);
12
13          shared_ptr(ControlBlock<T>* cb) : inner_block(cb), ptr(cb->val){};
14      public:
15          explicit shared_ptr(T* pointer){
16              inner_block = new ControlBlock<T>{1, pointer};
17              ptr = pointer;
18              if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>)
                    {
```

```
19                  ptr->wptr = *this;
20              }
21          }
22
23          shared_ptr(const shared_ptr<T>& other) {
24              ptr = other.ptr;
25              ++inner_block->shared_cnt;
26          }
27
28          shared_ptr(shared_ptr<T>&& other) {
29              inner_block = std::move(other.inner_block);
30              other.ptr = nullptr;
31          }
32
33          shared_ptr<T>& operator=(const shared_ptr<T>& other) & {
34              shared_ptr<T> copy(other);
35              std::swap(copy, *this);
36              return *this;
37          }
38          shared_ptr<T>& operator=(shared_ptr<T>&& other) & noexcept {
39              inner_block = std::move(other.inner_block);
40              other.ptr = nullptr;
41              return *this;
42          }
43
44          ~shared_ptr() {
45              if (!inner_block) return; // no control block
46              --inner_block->shared_cnt;
47              if (inner_block->shared_cnt == 0) {
48                  delete ptr; // obj deleted, but not control block
49                  if (inner_block->weak_cnt == 0 ) { // if false, then some
                        weak_ptr are looking at obj
50                      delete inner_block;
51                  }
52              }
53          }
54
55      };
```

14.3 make_shared

А еще плохой стиль написания кода - это мешать C-style указатели и умные указатели, так как почти во всех ситуациях будет UB.

```
1   std::shared_ptr<int>(new int(6));//bad code-style
```

Пусть f - функция которая кидает исключение. Мы не знаем в каком порядке компилятор обрабатывает аргументы, следовательно может так получиться, что он выделит память, вызовет функцию f и соответственное не успеет обернуть выделенную память в shared_ptr. Пример:

```
1   h(shared_ptr<int>(new int(6)), f(0));
```

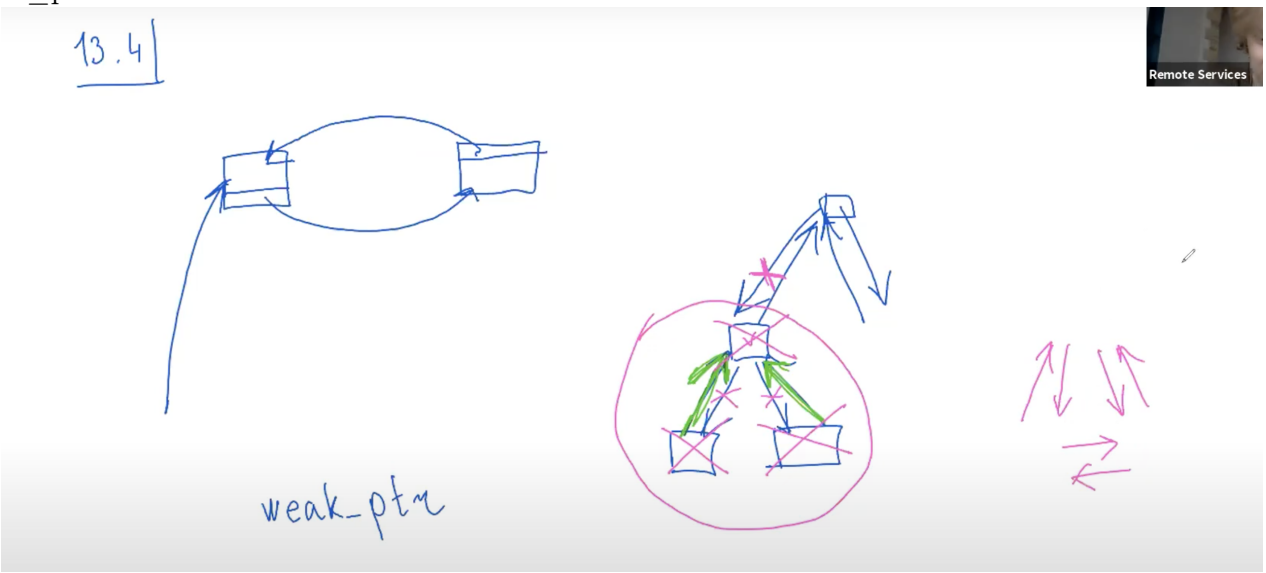
Если что, в C++17 это поправили, и теперь компилятор не имеет права вычислять аргументы функции в произвольном порядке

Для решения проблем с ручным выделением памяти придумали функции make_shared, make_unique(с C++14), allocate_ shared, allocate_unique(allocate делает все тоже самое что и make но с нестандартным аллокатором) Она принимает аргументы конструктора T и сама вызвает new от нужного типа и оборачивает в shared_ptr (создание обхекта вместе с контрольным блоком для него)

```
1     template <typename ...Args>
2     unique_ptr<T> make_unique(Args&& ...args) {
3         return unique_ptr<T>(new T(std::forward<Args>(args)...));
4     }
5
6
7     template <typename T, typename ...Args>
8     shared_ptr<T> make_shared(Args&& ...args) {
9         auto p = new ControlBlock<T>(1, std::forward<Args>(args)...);
10        return p;
11    }
12
13    int main(){
14        auto p = std::make_unique<int>(5); //copy elision called, no copy-
              ctor here
15    }
```

14.4 std::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, формульно будут оставаться объекты указывающие на вершинку - вершинка не удалится, и так со всеми вершинками в удаляемом дереве. Т.о. произошла утечка памяти. Решение пррблемы - weak_ptr



Это сущность, которая как и shared_ptr хранит указатель на некий объект, однако он им не вла-

деет (просто смотрит) . его можно спросить две вещи: не умер ли еще объект на который ты указываешь, создай новый shared_ptr на обхект, который ты смотришь для работы. Сам weak_ptr разыменовывать нельзя. Если объект убит, а мы попросили shared_ptr то это будет UB/RE. Weak_ptr ничего не делает с объектом, просто смотрит. Как это решает проблему - вершина понимает что на нее указывает 0 shared_ptr и 2 weak_ptr. но weak_ptrы не считаются - объект умирает

Правило: если есть двустороняя циклическая зависмость, то хотя бы один указатель (связь) в кадждой из стороны в ней должен быть вик птром. тогда если хотя бы одна связь из цикла будет разрушен, будет разрушен и весь цикл, причем в правильнрм порядке

Реализация

```
1     template<typename T>
2     class weak_ptr{
3     private:
4             ControlBlock<T>* inner_block = nullptr;
5     public:
6             weak_ptr(const shared_ptr<T>& p): inner_block(p.inner_block){};
7
8             bool expired() const {
9                 return inner_block->shared_cnt == 0;
10            }
11
12            shared_ptr<T> lock() const {
13                if (expired()) {
14                    throw std::bad_weak_ptr();
15                }
16                return shared_ptr<T>(inner_block);
17
18            }
19
20            ~weak_ptr() {
21                if(!inner_block) return;
22                --inner_block->weak_cnt;
23                if (inner_block->weak_cnt == 0 and inner_block->shared_cnt
                      == 0) {
24                    delete inner_block;
25                }
26            }
27    };
```

14.5 enable_shared_from_this

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

```
1     template<typename  T>
2     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>.

Замечание: попытка вызвать shared_from_this в случае, когда объектом не владеет ни один shared_ptr является UB(с C++17 выкидывается исключение std::bad_weak_ptr.

14.6 Умный указатель в качестве возвращаемого значения

Еще лучше, если функция func() явно будет возвращать умный указатель вместо обычного:

```
    std::unique_ptr<T> func() {
        return std::unique_ptr<T>(new T);
    }
    std::shared_ptr<T> g() {
        return std::make_shared<T>();
    }

    void anotherFunc() {
        std::unique_ptr<T> c = func();
        std::shared_ptr<T> c2 = func(); //OK
        // std::unique_ptr<T> c2 = g(); WRONG
        //std::shared_ptr<T> c3 = c; //still forbidden
    }
```

Это обяжет использовать безопасную конструкцию всех пользователей вашей функции.

14.7 Умные указатели для массивов

Оба класса умных указателей можно использовать для управления массивами:

```
    std::shared_ptr<int> a1(new int[10], std::default_delete<int[]>());
    std::unique_ptr<int> a2(new int[10]);
```

Заметим, что при создании shared_ptr потребовался дополнительный аргумент std::default_delete<int[]>(). Он необходим для корректного освобождения ресурсов, обеспечивая вызов delete[].Еще одно отличие - с c++17 определен оператор квадратные скобки.

15 Лямбда-функции и элементы функционального программирования

С C++11 появились лямбда-функции. Они помогают описывать функции прямо внутри выражения, которое их вызывает.

15.1 Идея и синтаксис

Выражение стоящее после [.] называется **замыканием** (closure). Оно является rvalue

Синтаксис

```
1    std::vector<int> v = {1,6,4,6,3,6};
2    std::sort(v.begin(), v.end(), [](int x, int y) {
3        return std::abs(x - 5) < std::abs(y - 5);
4    });
```

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)

```
1    auto getCompare() {
2        return [](int x, int y) {
3            return std::abs(x - 5) < std::abs(y - 5);
4        }
5    }
6
7    [](int x) {
8        std::cout << x << "\n"; //declaration, rvalue
9    };
10
11   [](int x) {
12       std::cout << x << "\n"; //called
13   }(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

```
1    [](int x) -> bool {
2        std::cout << x << "\n"; //declaration, rvalue
3    };
```

15.2 Захват

В теле лямбда-функции по умолчанию не видны локальные переменные внешней функции, если только она не передана как параметр. В ней видны глобальные переменные и переменные из пространств имен. Переменную, указываемую в [.] называют **захваченной**

```
1    int a = 6;
2    [a](int x) {
```

```
3         std::cout << x + a << "\n";
4     };
```

Capture list - список переменных внутри [.] Но что будет, если мы захотим поменять a?

```
1   //                                          CE
2       [a, b](int x)              {
3           ++a;
4           cout << x + a + b << endl;
5       }
```

По умолчанию, когда вы захватываете в лямбда-функцию переменную, она считается константной. Но есть способ этого избежать, нужно раскоммитить mutable (подробнее о нём было выше).

При захвате на переменные навешивается const "справа"на переменные, а значит, ссылка не становится константной. Следовательно, этот код будет менять переменную a:

```
1     [&a, b](int x) mutable {
2         ++a;
3         cout << x + a + b << endl;
4     }
```

Мотивация списков захвата: поскольку мы часто используем лямбда для передачи в качестве функтора, ожидается, что она принимает фиксированное количество параметров.

Захватывать по ссылке хорошо, так как нет варианта, когда функция "переживает"объект, ссылку которого она захватила.

15.3 Лямбда функции как объект

У лямбд свой собственный, обычно уникальный для каждой функции, тип. Размер лямбды равен 1 байту. Считается пустым функциональным классом с оператором круглые скобки.

Что произойдёт, если мы захватим перменную? **В этот класс добавится поле.** Оператор круглые скобки по стандарту является константным. Отсюда объяснение работы со ссылками: если поле класса есть ссылка, то по ней можно менять из конст-метода (см. странная особенность константных ссылок).

Оператор присваивания не генерируется для лямбды. С C++20 есть конструктор по умолчанию: decltype(f) ff;

15.4 Особенности захвата полей класса и this с помощью лямбды

Этот код не сработает, потому что поля нельзя захватывать.

```
1     struct S {
2         int a = 1;
3         void foo() {
4             auto f = [a](int x, int y) {
5                 cout << a;
6                 return x < y;
7             };
8         }
9     };
10
11    int main() {
```

```
12          S s;
13          s.foo();
14      }
```

Можно поправить:

```
1       struct S {
2           int a = 1;
3           void foo() {
4               auto f = [this](int x, int y) {
5                   cout << a;
6                   return x < y;
7               };
8           }
9       };
10
11      int main() {
12          S s;
13          s.foo();
14      }
```

15.5 Захват с инициализацией

Но что делать, если мы хотим захватить по значению? Практически важная проблема.

```
1       struct S {
2           int a = 1;
3           void foo() {
4           //make b and initialize it with a
5               auto f = [b = a](int x, int y) {
6                   cout << a;
7                   return x < y;
8               };
9           }
10      };
11
12      int main() {
13          S s;
14          s.foo();
15      }
```

Как следствие, мы можем мувать в лямбду: b = std::move(a).

Есть синтаксис захвата всех локальных переменных: [=], так же [&, s] - захвати всё по ссылке, кроме s. Так же они бывают обобщёнными (тогда внутри сгенерируется шаблонный класс):

```
1           void foo() {
2               auto f = [](auto x) {
3                   return x < y;
4               };
5           }
```

15.6 std::function

Тип, который позволяет вам инициализировать любым объектом, который является callable. Представьте, чтобы

```
1     struct S {
2     bool operator()(int x, int y) const {
3         return x < y;
4     };
5
6     bool g(int x, int y) {
7         return x < y;
8     }
9
10    int main() {
11        std::function<bool(int, int)> f;
12        //can assign lambda function
13        f = [](int x, int y) {
14            return x < y;
15        }
16        //or operator() of some class
17        f = S();
18        //or C-style function
19        f = &g;
20    }
```

В последнем случае & можно не писать, это синтаксический сахар. В аргументах и возвращаемом значении действует приведение типов. В моментах когда происходит подмена одного значения function на другое, прошлый объект удаляется.

У std::function есть конструктор копирования, перемещения, оператор присваивания и т.д - полноценный объект, которые можно передавать в другие функции по ссылке, значению и т.д

16 Шаблонное метапрограммирование

Шаблонное метапрограммирование - раздел программирования, в котором работа происходит не над объектами, а над типами

16.1 SFINAE

SFINAE - правило вывода шаблонных версий функций.
Substituon Failure Is Not An Error - неудачная шаблонная подстановка не является ошибкой компиляции.

Важно, что неудачная шаблонная подстановка должна происходить в момент инстанцирования сигнатуры(объявления) функции,а не при выборе специализации шаблонов или в теле функции

```
1     template <typename T>
2     auto f(const T&) -> decltype(T().size()) {
3         std::cout << 1;
4     }
5
6     //template <typename T>
7     //int f(const T&) {
8     //    T x;
9     //    x.size();
10    //}
11
12    void f(...) {
13        std::cout << 2;
14    }
15
16    int main() {
17        f(5); //2 is printed
18    }
```

Позволяет включать и выключать некоторые версии перегрузки функций в зависимости от некоторый компайл тайм проверяемых условий

Некоторая версия перегрузки работала тоьлко в случае если тип обладает какими-то определенными свойствами - например, тип является классом (есть type trait is_class)

Есть два вида мета-функций, которые возвращают тип (::type) и возвращают value(::value)

16.2 enable_if

Реализация + выбор перегрузки функции в зависимости от того, является ли T классом

```
1     template <bool B, typename T = void>
2         struct enable_if{};
3
4     template <typename T>
5     struct enable_if<true, T>{
6         using type = T;
7     };
8
9     template <bool B, typename T = void>
```

```
10      struct enable_if_t{
11          using type = typename enable_if<B, T>::type;
12      };
13      /////USAGE////
14
15      template<typename  T, typename std::enable_if_t<std::is_class_v<T>>>
16      void g(const T&) {
17              std::cout << 1;
18      }
19
20      template<typename T, typename = std::enable_if_t<!std::is_class_v<
21      std::remove_reference_t<T>>>>
22      void g(T&&) {
23              std::cout << 2;
24      }
```

16.3 has_method

Используется в Allocator_traits(присутствует ли метод construct) и Iterator_traits (какая категория итератора - проверить наличие всех качеств, которые есть например у RAI)

Пусть T - тип у которого проверяем наличие метода, Args - аргументы от которых должен быть проверяемый метод

Какие проблемы возникали при реализации has_method:

- Мы хотели, чтобы компилятор вычислял значение U().construct(UArgs()...), чтобы использовать SFINAE, однако при этом, чтобы применить is_same_v нужно явно знать вовращаемый тип этого выражения. Выход - оператор запятая, которая вычисляет оба выражения и возвращает правый.
- Функции должны быть static, так как мы не хотим ради проверки создавать объект has_method
- Перегруженные функции должны были возвращать значения разных типов чтобы использовать is_same_v
- Вспоминаем проблему с пушбеком в векторе - там происходит инстанцирование шаблонных параметров при инстанцировании класса, поэтому вызывать функцию f от тех же параметров что и сама структура нельзя - SFINAE не сработает, так как подстановка уже произошла. Выход - у функции должны быть свои шаблонные параметры
- Еще проблема -если нет конструктора по умолчанию, возможно иногда будет false вместо true(например конструктор по умолчанию удален, и выражение не может вызваться

Реализация

```
1       template<typename T, typename... Args>
2       struct has_method_construct {
3       private:
4           template<typename U, typename... UArgs>
5           static auto f(int) -> decltype(
6           declval<U>().construct(declval<UArgs>()...), int()
7           ){};
```

```
8
9           template<typename...>
10          static char f(...){};
11      public:
12          static const bool value = std::is_same_v<
13          decltype(f<T, Args...>(0)), int>;
14      };
15
16      struct S {
17          void construct(int) {}
18          void construct(int, double) {}
19      };
20
21      template<typename T, typename... Args>
22      const bool has_method_construct_v = has_method_construct
23      <T, Args...>::value;
```

Решение последней проблемы - функция declval() - используется когда нужен объект типа T, но мы не знаем есть ли конструктор по умолчанию. Ее не надо реализовывать - она не предназчначена для вызова непосредственно в констекстве выполнения, достаточно для обращения к себе под unevaluated context (под decltype, size, noexcept и т.д, в остальных случаях нельзя)

```
1       template<typename T>
2       T&& declval() noexcept;
```

&& дает преимущство в виде возможности работать с типами у которых нет тела (incomplete types - ссылку на такие создать можно) и с шаблонными типами, которые не придется инстанцировать до этого. declval по типу дает выражение этого типа, т.е в какой-то степени это противоположность decltype

16.4 is_constructible

Проверяет правда ли у типа U есть конструктор от заданных аргументов

Реализация

```
1       template<typename T, typename... Args>
2       struct is_constructible {
3       private:
4           template<typename U, typename... UArgs>
5           static auto f(int) -> decltype(
6           U(declval<UArgs>()...), int());
7
8           template<typename...>
9           static char f(...);
10      public:
11          static const bool value = std::is_same_v<
12          decltype(f<T, Args...>(0)), int>;
13      };
14      template<typename T, typename... Args>
15      const bool is_constructible_v = is_constructible
16      <T, Args...>::value;
```

16.5 is_copy_constructible, is_move_constructible

Реализация is_copy_constructible

```
template<typename T>
struct is_copy_constructible {
private:
    template<typename U>
    static auto f(int) -> decltype(
    U(declval<const U&>())), int());
declval
    template<typename...>
    static char f(...);
public:
    static const bool value = std::is_same_v<
    decltype(f<T>(0)), int>;
};
template<typename T>
const bool is_copy_constructible_v = is_copy_constructible
<T>::value;
```

Реализация is_move_constructible Проверяет, правда ли что можно сконструироваться от rvalue, поэтому && можно не навешивать дополнительно. Реализация аналогична is_copy_constructible, но в (1) declval<U>

Реализация без копипасты:

```
const bool is_copy_constructible_v = is_constructible
<T, const T&>::value;
const bool is_move_constructible_v = is_constructible
<T, T&&>::value;
```

16.6 is_nothrow_move_constructable

В 12.9 мы писали реализацию move_if_noexcept на основе std::if_move_constructible. Теперь реализация своего if_move_constructible.

Это проверка, что мув-конструктор не кидает исключений

Почему нельзя наивно выразить is_nothrow_move_constructable через std::if_move_constructible_v noexcept(T(std::declval<T>()))? Потому что правило игнорирования второго аргумента в конъюнции при ложном первом аргументе - это рантайм-действие, в Compile-Time все равно все скомпилируется, и упадет с CE

```
template<typename T>
auto move_if_noexcept(T& x) -> std::conditional<std::
    is_move_constructible_v<T> && noexcept(T(std::declval<T>())),
T&&, const T&> {
    return std::move(x);
}
```

Реализация

В STL есть такой мета-класс integral-consant -константа врмени компиляции

```
template<typename T, T v>
struct integral_constant {
```

```
3         static const T value = v;
4     }
5
6     struct true_type: integral_constant<bool, true>{};
7
8     struct false_type: integral_constant<bool, false>{};
9
10    template<typename T>
11    struct is_nothrow_move_constructible {
12    private:
13        //static auto f(int) -> //std::conditional_t<noexcept(
14        //std::declval<U>), true_type, //false_type>{}; //bad code-style
15
16        template<typename U>
17        static auto f(int) -> integral_constant<bool, noexcept(U(std::
              declval<U>()))>{};
18
19        template<typename...>
20        static auto f(...) -> false_type;
21    public:
22        static const bool value =
23        decltype(f<T>(0))::value;(1)
24    };
25
26    template<typename T>
27    const bool is_nothrow_move_constructible_v =
          is_nothrow_move_constructible
28    <T>::value;
```

(1) станет истинной тогда и только тогда, когда у типа есть move-constructor и он безопасен относительно исключений

16.7 is_base_of

Хотим проверить, является ли один класс наследником другого - то есть проверить можно ли подставить наследника вместо родителя. Так как скастовать указатель на один класс к указателю на другой класс можно только если первый класс является наследником второго - опять можем применить SFINAE

Проблемы с которыми столкнулись при реализации:

- Если наследование приватное, то будет CE, хотя должно говорить да. Выход - реализация функции test(), которая проверяет получилось ли вызвать функцию f. Если он свалился, то значит он попал в приватное наследование и надо вернуть true

```
1     template<typename T, T v>
2     struct integral_constant {
3         static const T value = v;
4     };
5
6     struct true_type: integral_constant<bool, true>{};
7
8     struct false_type: integral_constant<bool, false>{};
```

```
9
10      namespace detail_is_base_of {
11          template<typename B>
12          auto f(B*) -> true_type;
13
14          template<typename...>
15          auto f(...) -> false_type;
16
17          template<typename B, typename D>
18          auto test(int) -> decltype(detail_is_base_of::f<B>(std::declval<
                D*>()));
19
20          template<typename B>
21          auto test(...) -> true_type; //if private
22      }
23
24      template<typename B, typename D>
25      struct is_base_of: integral_constant<bool,
26          std::is_class_v<B> && std::is_class_v<D> &&
27          decltype(detail_is_base_of::test<B, D>(0))::value> {};
28
29      template<typename B, typename D>
30      const bool is_base_of_v = is_base_of<B, D>::value;
31
32      struct Base{};
33
34      struct Derived : Base{};
35
36      int main(){
37          std::cout << is_base_of_v<Base, Derived>;
38          std::cout << is_base_of_v<Derived, Base>;
```