- C++
  - Операции
  - Функции
    - Операторы
    - Lambda
    - Функторы
    - Пользовательские литералы
  - Типы данных
    - Преобразования типов
    - Динамическая типизация
      - RTTI
  - Указатели и ссылки
    - Модель памяти и время жизни
    - Сырые указатели
    - Умные указатели
  - Структуры и классы: ООП
    - Конструкторы и деструкторы
    - Статический полиморфизм
      - Перегрузка методов
    - Динамический полиморфизм
      - Виртуальные методы
      - Таблица виртуальных методов
      - Вызов виртуальных методов из конструктора и деструктора
      - Вызов чисто виртуальной функции
    - Правило 0\5
    - Множественное наследование
  - Исключения
    - Access violation
  - Типизация
    - const
    - volatile
    - auto\decltype
    - type-casting
  - Шаблоны
    - Вариативные шаблоны
    - Специализации шаблонов
      - Частичная специализация
      - Полная специализация
    - SFINAE
    - type-traits
  - Особенности языка
    - ADL
    - Global init fiasco
  - Стандартная библиотека
    - Потоки ввода вывода
    - Дата и время

        - Контейнеры
        - Итераторы
        - Алгоритмы
        - Многопоточность
    - Неопределенное и неуточненное поведение
    - C++20
    - Идеомы
        - RAII
        - pImpl
        - Non-copyable/non-movable
        - Erase-remove
        - Copy and swap
        - Copy on write
        - CRTP
    - Принципы разработки
        - SOLID
        - KISS
        - DRY
        - YANGI
        - BDUF
        - Композиция предпочтительней наследования
        - Бритва Окама
        - Разделяй и властвуй
    - Паттерны проектирования
    - Жизненный цикл ПО
    - Инструменты
        - Сборка - CMake
        - Разработка - VSCode
        - Отладка - gdb
        - Версирование - git
        - Линтер
        - Анализаторы
            - Статические
            - Динамические
    - Библоитеки
        - boost
        - google test\mock
    - Архитектура
    - Операционные системы
    - (Старое) Особенности языка
    - const
        - Переменные
        - Функции
    - Базовые
    - Последовательность
    - Запрещенные

C++

Операции

Операции приведены в порядке их приоритетности, каждая группа это отдельный уровень приоритета. Операции одного уровня имеют равных приоритет, который можно изменить при помощи скобок.

```
// :: Разрешения области видимости

// . и -> выбор члена класса
// [] индексация массива
// () вызов функции
// ++, -- постфиксные инкремент\декремент
// typeid механизм RTTI
// const_cast, dynamic_cast, reinterpret_cast, static_cast

// sizeof
// ++, -- префиксные инкремент\декремент
// ~ или compl число необходимое чтобы дополнить все разряды до 1
// ! или not логическое отрицание
// +, - унарные плюс и минус
// & операция взятия адреса
// * разименовывание указателя
// new, delete
// C-style cast ()

// .* и ->* разименовывание указателя члена класса

// *, /, % умножение, деление и остаток от деления

// +, - сложение и вычитание

// <<, >> побитовый сдвиг влево и вправо (*2, /2)
```

```
// <, >, <=, >= операции сравнения логических выражений

// ==, != или not_eq операции сравнения логических выражений

// & или bitand побитовое И

// ^ или xor побитовое исключающее И (1 если 0 и 1)

// | или bitor побитовое ИЛИ

// && или and логическое И

// || или or логическо ИЛИ

// ?: тернарный оператор
// = присвоение
// *=, /=, %=, +=, -=, <<=, >>=, &= или and_eq, |= или or_eq ^= or xor_eq
//throw

//, оператор запятая - вычисляет два значения и возвращает второе
c = (a, b); //c == b
c = a, b;   //c == a, вычисляется как (c = a), b
```

Функции

Операторы

Большинство операторов могут быть переопределены. Список не переопределяемых:

```
// . выбор члена класса
// .* выбор указателя на член класса
// :: разрешение области
// ?: тернарный оператор
// # препроцессор: преобразование в строку
// ## препроцессор: конкатенация
```

Операторы могут быть определены как член класса, или глобальная функция. Во втором случае функция принимает первым аргументом константную ссылку на объект. Рекомендации:

```
// Все унарные операторы - Член класса
// = () [] -> ->* - Обязательно член класса
// += -= /= *= ^= &= |= %= >>= <<= - Член класса
// Остальные бинарные операторы - Не член класса
```

Если оператор не является членом классом, он должен быть помечен как friend, для того чтобы работать с private\protected содержимым классов.

Примеры переопределений:

```
class A
{
    int a;

public:

    A(int a) : a(a) {}

    // Как глобальная функция
    friend const A operator+(const A& lhs, const A& rhs);

    // Как функция класса
    A operator-(A& other) {
        return A(a - other.a);
    }
};

const A operator+(const A& lhs, const A& rhs) {
    return A(lhs.a + rhs.b);
}
```

Перегрузка префиксных и постфиксных инкрементов и декриментов отличается тем, что постификсная форма принимает вторым аргументом int, который не используется

Lambda

Общий вид:

```
auto lambda = [capture-list](arguments) mutable
{
    ...
}; //Создание

//Если не указывать mutable - по дефолту аргументы не изменяемы

lambda(arguments); //Вызов
```

Списки захвата:

```
[] // ничего не захватывается
[=] // локальные переменные по значению
[&] // локальные переменные по ссылке
[this] // this по ссылке
[*this] // объект по копии, нужен mutable для вызова не const f()
[a, &b] // захват отдельных перменных, по значению и ссылке
```

```
[&r = x] // захват переименованной ссылки
[x = x + 1] // инициализация переменной, может быть std::move()
```

Как возвращаемое значение, так и аргументы могут быть типа auto.

Если необходимо хранить переменную или контейнер lambd:

```
std::function<return_type(arguments_types)> lamda;

std::vector<std::function<int(int)>> lambas_vector;
```

Существует возможность использовать обобщенные лямбды с переменным числом аргументов:

```
auto variadic_lambda = [](auto... args) { function(args...); }
// Perfect forwarding:
auto variadic_lambda_ = [](auto&&... args) { std::forward<decltype(args)>
(args)...; }
```

Так же они могут быть помечены как constexpr, если возможно вычисления будут выполненны на этапе компиляции.

Функторы

Функтор это объект, у которого перегружен оператор (). Они активно используются в STL, наравне с lambda, и например могут быть переданы в качестве аргумента в функцию сортировки.

Могут быть помечены как constexpr.

Так же существуют функторы из стандартной библиотеки:

```
// Арифметические
std::plus<int>{};
std::minus<int>{};
std::multiplies<int>{};
std::divides<int>{};
std::modulus<int>{};
std::negate<int>{};

// Логические
std::less<int>{}(1, 0);
std::greater<int>{}(1, 0);
std::equal_to<type>{};
std::not_equal_to<type>{};
std::greater_equal<type>{};
std::less_equal<type>{};

std::logical_and<type>{};
```

```
std::logical_or<type>{};
std::logical_not<type>{};

// Побитовые
std::bit_and<type>{};
std::bit_or<type>{};
std::bit_xor<type>{};
std::bit_not<type>{};

// Хэширование
std::hash<Arithmetic>{};
std::hash<Enum>{};
std::hash<std::nullptr_t>{};
std::hash<T*>{};

// Searchers
std::default_searcher<ForwardIt, BinaryPredicate>{};
std::boyer_moore_searcher<ForwardIt, BinaryPredicate>{};
std::boyer_moore_horspool_searcher<ForwardIt, BinaryPredicate>{};

// Отрицание функции (функция как аргумент)
std::not_fn<F>{};
```

Пользовательские литералы

Стандартные строковые литералы:

```
"Text" //char
L"Text" //wchar_t

u8"Text" //char - utf8
u"Text" //char16_t
U"Text"//char32_t

//Сырые строки обрамляются в () в "" и могут иметь произвольный delemiter
R"delimiter( raw string )delimeter"
LR"delimiter( raw string )delimeter"

u8R"delimiter( raw string )delimeter"
uR"delimiter( raw string )delimeter"
UR"delimiter( raw string )delimeter"
```

Помимо стандартных литеральных типов, можно определять пользовательские.

Пример пользовательского литерала преобразования радиан в градусы.

```
long double operator""_degrees(long double value)
{
    return value * M_PI / 180.0;
```

```
}

double degrees = 0.38__degrees
```

Список возможных аргументов, при определении пользовательского литерала:

```
( const char * )
( unsigned long long int )
( long double )
( char )
( wchar_t )
( char16_t )
( char32_t )
( const char * , std::size_t )
( const wchar_t * , std::size_t )
( const char16_t * , std::size_t )
( const char32_t * , std::size_t )
```

Типы данных

C++ является статически типизированным языком, это значит что тип переменной не может меняться после инициализации.

Преобразования типов

Безопасные преобразования, без потери данных:

```
bool -> char -> short -> int -> double -> long double
bool -> char -> short -> int -> long -> long long
unsigned char -> unsigned short -> unsigned int -> unsigned long
float -> double -> long double
```

Динамическая типизация

RTTI

Существует ключевое слово typeid, которое может принимать как аргумент тип или выражение.

```
typeid(int).name();
typeid(expression).nam();
```

Помимо получения имени типа есть ещё функции в типе std::type_info, возвращаемым оператором typeid().

Проверка является ли тип надлежащим.

```
typeid(int).before(typeid(char)); // == false
typeid(char).before(typeid(int)); // == true
```

Возможно получить хэш значение из std::type_info

```
size_t hash_id = typeid(int).hash_code();
```

Указатели и ссылки

Указатель ссылается на область в памяти, они могут быть взяты от некоторой перменной, могут быть сравненны, к ним пременимы операции сложения и вычитания, и они могут быть разименованы, для получения значения переменной на которую они указывают.

Ссылки похожи на указатели, но они не содержут операций сравнения и операции сложения и вычитания.

Модель памяти и время жизни

- Глобальны переменные
- Локальные переменные и агрументы функции, уничтожаемые при выходе из области видимости
- Динамически аллоцированные данные

Так же особые модели памяти используются в многопоточных приложениях, при взаимодействии с std::atomic, о них в разделе многопоточность.

Сырые указатели

Используются для предоставления доступа к объекту, через указатель на его базовый класс.

Так же при работе с сырыми указателями существует возможность конструирования объекта в уже выделенной памяти (placement new).

```
unsigned char bufer[sizeof(int)] ;

int *pInt = new (bufer) int(42);
```

При использовании placement new не вызывается delete, но требуется вызывать деструктор, если он есть у объекта.

```
unsigned char bufer[sizeof(NewClass)] ;
NewClass *pClass = new (bufer) NewClass(42);
pClass->~NewClass(); //Без этой строчки объект не будет разрушен
```

Умные указатели

- std::unique_ptr - основной тип умного указателя
- std::shared_ptr - указатель может использоваться в разных частях программы, деалокация происходит при уничтожении последнего объекта класса std::shared_ptr
- std::weak_ptr - используется для решения проблемы перекрестных ссылок, когда объекты ссылаются друг на друга, и не могут использовать shared_ptr, т.к. иначе память не будет высвобождена

std::weak_ptr создаётся из объекта std::shared_ptr, и может проверять является ли указатель висячим.

Умные указатели могут так же использоваться для аллокации массивов:

```
// Для того чтобы освобождать память правильно может понадобится свой deleter
std::shared_ptr<int> array_with_deleter(new int[10], std::default_delete<int[]>
());

//Эквивалентно:
std::shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });

//Однако если не требуется использование общей памяти можно использовать
std::unique_ptr
std::unique_ptr<int[]> unique_array(new int[10]);
```

Структуры и классы: ООП

Конструкторы и деструткоры

Конструкторы вызываются от базового класса, к дочерним. Деструкторы вызываются в противоположном направлении, от дочерних классов к базовому.

Если конструктор базового класса требует аргументов, тогда он должен быть вызван в конструкторе дочернего класса:

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

class B  : public A
{
public:
    B() : A(1) {}
};
```

Статический полиморфизм

Перегрузка методов

В языке C++ могут быть перегруженны функции. Перегруженные функции имеют одинаковое название, но разное количество аргументов, и\или разных тип аргументов.

Вызываемая функция определяется на этапе компиляции.

Динамический полиморфизм

Виртуальные методы

Виртуальные функции реализуют механизм позднего связавания, или дианамический полиморфизм. Он отличает от статического тем, что происходит на этапе выполнения.

Для этого требуется пометить функцию в базовом классе ключевым словом virtual. И определить такую же функцию в дочернем классе. При переопределении функции в дочернем классе крайне желательно использовать ключевое слово override, оно позволяет проще читать код, а так же может защитить от ошибки когда сигнатуры функций различались.

Суть полиморфизма реализуемого через виртуальные функции сводится к идее использования указателя\ссылки интерфейса (базового класса) для разных реализацией дочернего.

Таблица виртуальных методов

Указатель на таблицу виртуальных методов vptr присутствует в любом классе, которых содержит хотя бы 1 виртуальную функцию.

Если в базовом классе есть хотя бы 1 виртуальная функция, деструктор в ней должен быть помечен virtual. В противном случае при удалении указателя на базовый класс, по адресу которого находится дочерний класс - будет вызван только деструктор базового класса.

При наличии аргументов по умолчанию, будут выбираться те, что соответствуют типу указателя, по которому вызывается функция объекта. Т.е. это происходит на этапе компиляции, и для родительского указателя будет взят родительский аргумент по умолчанию, а для дочернего - дочерний элемент по умолчанию.

Вызов виртуальных методов из конструктора и деструктора

Когда виртуальная функция вызывается из конструктора, т.к. конструкторы дочерних классов ещё не вызывались и потому реализованные в них функции не будут вызванны. Таким образом при вызове виртуальной функции из конструктора будет вызванна та его версия, которая заполненна в таблице виртуальных функций к моменту выполнения конструктора.

При вызове виртуальных функций из деструктора возникает схожая проблема. Т.е. если мы вызываем виртуальную функцию из деструктора базового класса, будет вызванна та его версия, которая находится в таблице виртуальных методов. Те деструкторы дочерних классов, которые уже отработали, очистят таблицу виртуальных функций от своих методов.

Однако если поведение описанное выше соответствует задуманному, т.е. ошибка отсутствует можно вызывать функции используя явное указание на класс из которого вызывается функция. Это никак не меняет поведение программы, но читать такой код проще, и компилятор\статический анализатор не выдадут предупреждения о возможной ошибке.

```
B() {
  std::cout << "B()\n";
  A::foo();
  B::bar();
};
```

Вызов чисто виртуальной функции

Чисто виртуальные функции используются в базовых классов, в которых отсутствует реализация этого виртуального метода. Подобная методология позволяет создавать абстрактные классы, объекты которых нельзя создать. Однако указатели на такие классы позволяют обращаться к дочерним классам и вызывать их виртуальные функции.

Если вызвать такую функцию из конструктора базового класса это приведёт к аварийному завершению программы.

Правило 0\5

Правило 3 и 5 гласит, что если нам требуется реализация копирующего или перемещающего конструкторов, или соотвествующих операторо, или деструктора работающего с выделяемой памятью - рекомендуется реализовать все эти функции.

Правило 0 гласит, что не следует вручную создавать такие функции, а доверить им поведение по умолчанию = default. А для хранения указателей использовать их умные версии, избавляя себя от необходимости реализовывать деструктор.

Множественное наследование

Исключения

Access violation

Типизация

const

volatile

auto\decltype

type-casting

Шаблоны

Вариативные шаблоны

Специализации шаблонов

Частичная специализация

Полная специализация

SFINAE

Очень коротко!

type-traits

Очень коротко с ссылками!

Особенности языка

ADL

Global init fiasco

+???

Стандартная библиотека

Потоки ввода вывода

Дата и время

Контейнеры

Итераторы

Алгоритмы

Многопоточность

Неопределенное и неуточненное поведение

https://en.cppreference.com/w/cpp/language/eval_order

C++20

Идеомы

RAII

pImpl

Non-copyable/non-movable

Erase-remove

Copy and swap

Copy on write

CRTP

+???

Принципы разработки

SOLID

KISS

DRY

YANGI

BDUF

Композиция предпочтительней наследования

Бритва Окама

Разделяй и властвуй

Паттерны проектирования

Жизненный цикл ПО

Инструменты

Сборка - CMake

Разработка - VSCode

Отладка - gdb

Версирование - git

Линтер

Анализаторы

Статические

Динамические

Библоитеки

boost

google test\mock

Архитектура

Операционные системы

(Старое) Особенности языка

const

Переменные

```
//Константный int
const int i = 1;

//Альтернативная запись
int const j = 2;

//Указатель на константный int
const int* pI = &i;

//Константный указатель
int const* const cP = &i;

//Снятие модификатора
const_cast<int&>(i) = 3;

int k = 0;
// Ошибка компиляции, так можно добавить константность
static_cast<const int&>(k) = 1;
```

Функции

```
class A
{
    std::string value = "test";
    int i = 0;

    mutable int j = 0;

    int* p_i; //Предположим что оно проинициализированно

public:

    void f(const int i) {}

    //Ошибка компиляции, сигнатура считается одинаковой
    void f(int) {}

    //Так функции могут быть перегружены
    void f_ref(int& r_i) {}

    //Данная функция будет вызванна от r-value или const int как аргумента
    void f_ref(const int& c_r_i) {}

    //Возврат константной ссылки
    const string& get_const_ref() { return value; }

    //const функция не даст изменить состояние объекта
    //Из неё могут вызываться только другие const функции
    int const_f() const { return i; }

    //const функции можно перегружать, и они будут выбираться в зависимости от
нашего объекта типа A
    int const_f() { return i };

    //Модификация j разрешена из-за ключевого слова mutable
    void set_const_f() const { j = 1; }


    //Такой вызов разрешён, т.к. сам указатель не меняется
    void update_value() const { *p_i = 0; }

    //Другой метод обойти константность, помимо mutable
    void trick() const
    {
        const_cast<A*>(this)->i = 1;
    }
}
```

#Конструкторы и деструкторы

Базовые

```
class A
{
    std::string str;
    std::vector vec;

public:

    //Конструктор по умолчанию, конструирует str\vec по умолчанию
    A() {}

    //Разрушает str\vec
    ~A() {}

    //Конструктор копирования - может быть сгенерирован, если объект может
копироваться
    //Например наличие ссылки в членах класса не даст ему сгенерироваться
автоматически
    A(const A& other) {}

    //Конструктор перемещения
    A(A&& other) {}


    //Если требуется только стандартное поведение:
    A() = default;

    //Инициализация в констркторе будет происходить в последовательности членов в
классе, т.е. str\vec
    A() : vec({0}), str("hi") {}

};
```

Последовательность

Конструкторы и деструкторы в одном объекте. Конструирование и разрушение из одного скопа. (Принцип один)

Запрещенные

Закрытые конструкторы\деструкторы:

```
class A
{
    //Старый метод запретить
private:
    ~A() {}
    A() {}

public:
    //Современный метод
```

```
    ~A() = deleted;
    A() = deleted;
};
```

Так можно запретить конструктор копирования или перемещения. Конструктор можно запретить, например, в синглетоне.

Если конструктор запрещён, то объект можно создавать в статических функциях или классах\функциях friend.

Если деструктор удален есть способ удалить объект:

```
class A
{

public:

    ~A() = deleted;

    void destroy() { delete this; }
};

Однако такой объект может существовать только в динамической памяти, создать его
на стеке не удастся, т.к. он не способен потом разрушится.


## Виртуальный деструктор

```cpp
class A
{
public:

    virtual ~A() {}

    virtual f() { std::cout << "A::f" << endl; }

};

class B : public A
{
public:
    ~B() {}

    virtual f() { std::cout << "B::f" << endl; }
};


A* a = new B();
//Если не сделать деструктор виртуальным будет вызван только ~A
delete a;
```
```

```
//Есть возможность достичь результата без виртуального деструктора - если создать объект shared_ptr<B> и вернуть его как shared_ptr<A> он будет разрушен в деструкторе shared_ptr, но с unique_ptr не выйдет
```

Все классы в STL без виртуального деструктора, потому надо быть осторожным с ними, если возникнет потребность их наследовать.

Исключения в деструкторе

Исключение выброшенное из деструктора наружу может скрашить приложение (terminate). Причём один блок try может обработать только 1 исключение, т.е. если обернуть в try создание двух объектов, и оба выкинут исключения - программа завершится аварийно.

Одно из решений, это обрабатывать исключения внутри деструктора.

Так же может быть хорошей идеей вынести код, который может сгенерировать исключение - в отдельную функцию, и вызвать её явно, в блоке try.

```

```

Виртуальные функции в конструкторе или деструкторе

```
class A
{
public:

    A() { f(); }
    ~A() { f(); }

    virtual f() { std::cout << "A" << endl; }
};

class B : public A
{
public:

    B() {}

    virtual f() { std::cout << "B" << endl; }
};


//Эта строчка выведет A
B b;

//Эта строчка выведет B
```

```
b.f();

//При разрушении B будет выведено тоже A
```

Нужно избегать вызова виртуальных функций, в конструкторе или деструкторе.

Виртуальный "конструктор"

Рассмотрим проблемную ситуацию

```
class A {

public:
    virtual A* clone() { return (new A(*this)); }

};

class B : public A {

public:

    //co-variant return type: перегрузка функции возвращающей разные указатели
    virtual B* clone() { return (new B(*this)); }

};

void foo(A* to_copy)
{
    //Хоть мы передаём объект B, сконструирован по копии будет A
    A* a = new A(*to_copy);

    Решение
    A* a2 = to_copy->clone();
}

B b;
foo(&b);
```

Возможные ошибки

Оператор присвоения

Для оператора присвоения надо проверять, что объект не присваивается самому себе.

Static intialization fiasco

Проблема возникает при инциализации одной глобальной переменной другой. Этого надо строго избегаить.

Преобразования типов

Неявное\неявное преобразование типов

```
class A {};

class A2 {};

class B
{
    std::string str = "test";

public:

    B(const A& a) {}

    explicit B(const A2& a) {}

    std::string operator() const { return str; }
};

A a;
B b = a; //Неявное преобразование

A2 a2;
B b2 = a2; //Запрещено

std::string implicit = b; //Неявное преобразование
```

Опасной может быть ситуация, когда не explicit конструктор имеет 2 аргумента, со дефолтными значениями, и объект такого типа создается присвоением одной переменной соответствующего типа. Будет неявное преобразование типа, с поведением, которое не задуманно.

cast

😃

Идеомы \ техники

Статический полиморфизм

CRTP

Множественное наследование

Ромбовидное

Argument Dependent Lookup (ADL)

namespace lookup

Многопоточность