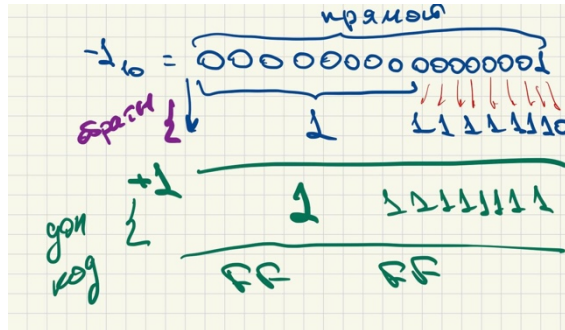


- Ответ:** FF FF тк это дополнительный код



- Ответ:**

- Эти значения могут быть изменены при компоновке программы параметрами компоновки.

```
/STACK:reserve[,commit]
```

[/STACK:4194304](#)

- **Linux:** Размер стека для новых потоков в Linux составляет **8 МБ** для 32-битных и 64-битных приложений. POSIX threads.

В GCC размер стека по умолчанию определяется операционной системой, и его непосредственно **через компоновщик изменить нельзя**. Однако, можно управлять размером стека для новых потоков, используя функцию `pthread_attr_setstacksize` при работе с потоками POSIX.

Для изменения размера **стека главного потока программы** на уровне исходного кода, можно использовать функцию `setrlimit` из библиотеки `<sys/resource.h>`, например:

```
#include <sys/resource.h>

// изменение размера стека для главного потока

int main() {
    const rlim_t kStackSize = 16 * 1024 * 1024; // желаемый
    размер стека в байтах
    struct rlimit rl;
    int result;

    result = getrlimit(RLIMIT_STACK, &rl);
    if (result == 0) {
        if (rl.rlim_cur < kStackSize) {
            rl.rlim_cur = kStackSize;
            result = setrlimit(RLIMIT_STACK, &rl);
            if (result != 0) {
                // Обработка ошибки
            }
        }
    }

    // Ваш код

    return 0;
}
```

Почему не безопасно изменять размер стека?

Ответ:

- **Переполнение стека:** рекурсия, выделение больших объемов данных на стеке. Переполнение стека может привести к **ub**, включая сбои и уязвимости безопасности.

- **Нехватка памяти:** аварийному завершению процессов из-за нехватки памяти. Тк другие процессы тоже используют стек.

- **Портируемость:** потребуется доп настройки для стороннего окружения.

- **Сложность отладки**

П4ут. предпочтительнее оптимизировать алгоритмы и структуры данных для уменьшения потребления стека, чем изменять его размер.

3) Что такое указатель и как он используется?

Ответ:

Указатель в С — это переменная, значение которой является адресом в памяти. используются для хранения адресов переменных, что позволяет прямо работать с памятью и изменять значения переменных

по их адресам. Для объявления указателя используется символ *, а для доступа к значению по адресу — операция разыменования *. Пример объявления указателя на int: `int *ptr;`. Для получения адреса переменной используется оператор &.

- 4) Как работает динамическое выделение памяти в С?

Ответ:

Динамическое выделение памяти в С осуществляется с помощью функций `malloc`, `calloc`, `realloc`, и `free`. `malloc` выделяет блок памяти заданного размера, `calloc` — аналогично, но дополнительно инициализирует память нулями, `realloc` изменяет размер уже выделенного блока, а `free` освобождает выделенную память. Важно всегда освобождать выделенную динамически память.

- 5) Каковы преимущества и недостатки использования глобальных переменных?

Ответ:

Плюсы:

- **удобство доступа** из любой части программы.

Недостатки:

- увеличение **зависимости** между модулями программы, затруднение отладки и тестирования из-за **возможности изменения значений**.

- 6) Что такое указатель на функцию и как его использовать?

Ответ:

Это указатель, который хранит адрес функции. Это позволяет динамически выбирать функцию для вызова. возвращаемый тип, за которым следует `(*имя_указателя)(параметры)`. Пример использования: `void (*fun_ptr)(int) = &myFunction;`

- 7) Как вы реализуете глубокое и поверхностное копирование структур?

Ответ:

Поверхностное копирование копирует значения структуры напрямую, включая адреса указателей, что может привести к разделению одного и того же участка памяти между двумя структурами. Глубокое копирование создает копию всех данных, на которые указывает структура, включая выделение новой памяти для копий всех указателей, чтобы каждая структура имела свою собственную копию данных.

- 8) Как в проект по С++ включить код на Си?

Ответ: `extern C`

Раздел C++

1) Умные указатели

Ответ:

Умные указатели были созданы для устранения вышеупомянутых неудобств. По сути, они обеспечивают **автоматическое управление памятью**: когда умный указатель больше не используется, **то есть выходит из области видимости**, память, на которую он указывает, **автоматически высвобождается**. Традиционные указатели теперь также называют «обычными» указателями.

Умный указатель **перегружает операторы -> и ***. Когда умный указатель выходит из области видимости, **срабатывает его деструктор** и происходит очистка памяти.

Виды:

- `std::unique_ptr` — умный указатель, владеющий динамически выделенным ресурсом;
- `std::shared_ptr` — умный указатель, владеющий разделяемым динамически выделенным ресурсом. Несколько `std::shared_ptr` могут владеть одним и тем же ресурсом, и **внутренний счетчик ведет их учет**;
- `std::weak_ptr` — подобен `std::shared_ptr`, но не увеличивает счетчик.

`std::unique_ptr`:

`std::unique_ptr` владеет объектом, на который он указывает, и никакие другие умные указатели не могут на него указывать. Когда `std::unique_ptr` выходит из области видимости, объект удаляется. Это полезно, когда вы работаете с временным, динамически выделенным ресурсом, который может быть уничтожен после выхода из области действия.

Лучше выделять объекты с помощью `make_unique`. Копия запрещена.

```
int main()
{
    std::unique_ptr<int[]> ptr = std::make_unique<int[]>(1024);
    std::unique_ptr<int[]> ptr_copy = ptr; // ОШИБКА! Копирование запрещено
    compute(ptr); // ОШИБКА! `ptr` передается копией, а копирование не
}
// разрешено
```

Это сделано специально, и это важная особенность `std::unique_ptr`: на любой ресурс может указывать **не более одного** `std::unique_ptr`. Это предотвращает **ошибочное многократное удаление указателя**. Тк нет **конструктора копирования**.

std::shared_ptr

`std::shared_ptr` владеет объектом, на который он указывает, но, в отличие от `std::unique_ptr`, он допускает **множественные ссылки**. Специальный **внутренний счетчик уменьшается** каждый раз, когда `std::shared_ptr`, указывающий на тот же ресурс, **выходит из области видимости**. Эта техника называется подсчетом ссылок. Когда **последняя из них будет уничтожена**, счетчик станет **равным нулю**, и данные **будут высвобождены**.

```
std::shared_ptr<int> p1 = std::make_shared<int>();
std::shared_ptr<Object> p2 = std::make_shared<Object>("Lamp");
```

До C++ 17 было **проблемой удалять ресурс-массив** тк по умолчанию был `delete p` (вместо `delete []`). Решение: **кастомный делитер**.

Один из многих конструкторов `std::shared_ptr` принимает в качестве второго параметра лямбду

```
std::shared_ptr<int[]> p2(new int[16], [] (int* i) {
    delete[] i; // Кастомное удаление
});
```

Но не при `make_shared`.

`Use_count()` для отслеживания кол-ва указателей, которые **ссылаются на один и тот же ресурс**

```
void compute()
{
    std::shared_ptr<int> ptr = std::make_shared<int>(100);
    // ptr.use_count() == 1
    std::shared_ptr<int> ptr_copy = ptr;    // Сделать копию: с shared_ptr
    // ptr.use_count() == 2
    // ptr_copy.use_count() == 2, в конце концов, это одни и те же базовые
    // данные.
} // Здесь `ptr` и `ptr_copy` выходят из области действия. Больше никаких
// исходные данные (т.е. use_count() == 0), поэтому они автоматически
// убираются.
int main()
{
    compute();
}
```

Кольцевые ссылки Логично, не так ли? К сожалению, я только что создал так называемую **круговую ссылку**. В начале моей программы я создаю два умных указателя `jasmine` и `albert`, которые хранят динамически создаваемые объекты: назовем эти динамические данные `jasmine-data` и `albert-data`, чтобы было понятнее.

Затем в (1) я передаю `jasmine` указатель на `albert-data`, а в (2) `albert` хранит указатель на `jasmine-data`. Это все равно что дать каждому игроку компаньона.

Когда `jasmine` выходит из области видимости в конце программы, ее деструктор не может очистить память: все еще есть один умный указатель, указывающий на `jasmine-data`, это `albert->companion`. Аналогично, когда `albert` выходит из области видимости в конце программы, его деструктор не может очистить память: ссылка на `albert-data` все еще живет через `jasmine->companion`. В этот момент программа просто завершается, не освободив память:

```

struct Player
{
    std::shared_ptr<Player> companion;
    ~Player() { std::cout << "~Player\n"; }
};

int main()
{
    std::shared_ptr<Player> jasmine = std::make_shared<Player>();
    std::shared_ptr<Player> albert = std::make_shared<Player>();

    jasmine->companion = albert; // (1)
    albert->companion = jasmine; // (2)
}

```

Если вы запустите приведенный выше фрагмент, то заметите, что `~Player()` никогда не будет вызван.

`std::weak_ptr`

`std::weak_ptr` — это, по сути, `std::shared_ptr`, который **не увеличивает счетчик ссылок**. Он определяется как умный указатель, который содержит несобственную ссылку, или **ослабленную ссылку**, на объект, управляемый другим `std::shared_ptr`.

Этот умный указатель полезен для решения некоторых раздражающих проблем, которые нельзя решить с помощью необработанных указателей. Вскоре мы увидим, как это сделать.

Вы можете создать `std::weak_ptr` только из `std::shared_ptr` или другого `std::weak_ptr`. Например:

```

std::shared_ptr<int> p_shared = std::make_shared<int>(100);
std::weak_ptr<int> p_weak1(p_shared);
std::weak_ptr<int> p_weak2(p_weak1);

```

`std::weak_ptr` является своего рода **инспектором** для `std::shared_ptr` от которого он зависит. Вы должны сначала преобразовать его в `std::shared_ptr` с помощью метода `lock()` если вы действительно хотите работать с реальным объектом:

```

std::shared_ptr<int> p_shared = std::make_shared<int>(100);
std::weak_ptr<int> p_weak(p_shared);
// ...
std::shared_ptr<int> p_shared_orig = p_weak.lock();

```

С помощью `std::weak_ptr` очень легко решить проблему **висящих указателей** — тех, которые указывают на уже удаленные данные. Он предоставляет метод `expired()`, который **проверяет, был ли объект**, на который ссылается ссылка, уже **удален**. Если `expired() == true`, исходный объект был где-то удален, и вы можете действовать соответствующим образом. Это то, что вы не можете сделать с необработанными указателями.

По скорости умные указатели работают **также**, как и обычные указатели, кроме `std::shared_ptr` из-за **подсчета ссылок**.

При использовании `make_unique()` и `make_shared()` код **защищен от исключений**.

1. Без объяснения как на практике устроен `std::shared_ptr`, невозможно понять что за сценой присутствует **control block** и без `std::make_shared` у нас появиться **дополнительный уровень косвенности** и следственно **лишнее выделение памяти под него**. `std::make_shared` объединяет эти два выделения, делая создание и удалении быстрее.
2. `std::weak_ptr` естественно делает **увеличение счетчика ссылок**, но это **отдельный счетчик**, который также находится в control block-е.

QT

Особенности QScopedPointer:

- **Принадлежность к библиотеке Qt:** `QScopedPointer` был разработан как часть библиотеки Qt и используется в коде, написанном с использованием этой библиотеки.
- **Простота:** `QScopedPointer` предоставляет базовые функции для управления памятью без возможности передачи владения, что делает его более простым в использовании для определенных случаев.
- **Ограниченная функциональность:** В отличие от `std::unique_ptr`, `QScopedPointer` не поддерживает некоторые расширенные возможности, такие как перенос владения (ownership transfer) или преобразование в другие типы умных указателей.

Особенности std::unique_ptr:

- **Часть стандартной библиотеки C++:** `std::unique_ptr` введен в C++11 и является частью стандартной библиотеки.
- **Переносимость и гибкость:** Поддерживает перенос владения с помощью семантики перемещения, что позволяет использовать его в более широком спектре сценариев, включая возвращение из функций и хранение в контейнерах.
- **Настраиваемый деструктор:** `std::unique_ptr` может быть настроен с использованием пользовательского деструктора, что делает его более гибким при работе с ресурсами, требующими специальной процедуры освобождения.
- **Более широкий набор функций:** Поддерживает различные операции, такие как сравнение, преобразование в `std::shared_ptr` и другие, что делает его более универсальным инструментом для управления ресурсами.

Сравнение:

Критерий	QScopedPointer	std::unique_ptr
Библиотека	Qt	Стандартная библиотека C++
Семантика перемещения	Не поддерживается	Поддерживается
Настраиваемый деструктор	Нет	Да
Перенос владения	Не поддерживается	Поддерживается
Использование в контейнерах	Ограничено	Полностью поддерживается

В целом, `std::unique_ptr` является более гибким и мощным инструментом для управления ресурсами по сравнению с `QScopedPointer`. Однако `QScopedPointer` может быть предпочтительнее в среде Qt для простых задач управления памятью, где не требуется перенос владения или настраиваемые деструкторы.

2) Что такое и зачем нужно Forward Declaration?

Ответ:

forward declaration означает декларирование идентификатора до того, как ему дано полное определение. Это требуется для того, чтобы компилятор знал тип идентификатора.

```
class MyClass;
struct MyStruct;
```

На языке C++ классы могут быть **forward-declared**, если только Вам нужно использовать тип указателя на этот класс (все указатели имеют одинаковый размер). Особенно это полезно внутри определений класса, например если класс содержит в себе член, который является указателем на другой класс; чтобы избежать циклических ссылок (например, этот класс может также содержать член, который указывает на этот же класс), мы просто вместо этого делаем предварительное декларирование классов.

Когда используем include, а когда **forward declaration**?

Forward declaration для класса недостаточно, если Вам нужно использовать **действительный тип класса**. Например, если мы имеем поле класса, который имеет **тип не указателя**, а **напрямую задает класс**, или если Вы используете его как **базовый класс**, или если Вам нужно использовать **методы этого класса в другом методе**.

Передавая в header include мы замедляем время компиляции.
Почему стоит использовать **forward-declaration**?

1. **Уменьшение времени компиляции:** Предварительное объявление классов и функций позволяет компилятору не загружать и не обрабатывать полные определения, что может существенно сократить время компиляции для больших проектов.
2. **Разрешение циклических зависимостей:** В ситуациях, когда два или более классов зависят друг от друга (например, когда объекты одного класса содержат указатели или ссылки на объекты другого класса и наоборот), предварительные объявления позволяют разорвать эти циклы, сообщая компилятору, что такие классы существуют, без необходимости предоставления полного определения до момента его использования.
3. **Избегание ненужных включений:** Использование предварительных объявлений вместо полных определений через заголовочные файлы помогает избежать излишних включений (**#include**), что также способствует уменьшению времени компиляции и предотвращает возможные проблемы с зависимостями и переполнением пространства имен.

- 3) Std контейнеры, как реализованы, что под капотом
Ответ:

std::array<T, N>

std::array<T,N>

std::array — это контейнер, который обеспечивает функциональность **статического массива** в C++. Входит в состав стандартной библиотеки шаблонов (STL) языка программирования C++ и доступен начиная с **стандарта C++11**.

std::array обладает возможностями контейнеров (итераторами), такими как **.size()**, **.begin()** **.end()**

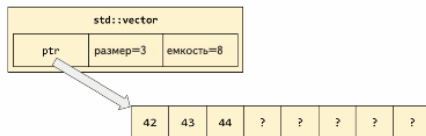
std::array хранит свои элементы в **статически выделенной** (на стеке)

std::vector<T>

std::vector представляет **непрерывный массив** элементов данных, но место для него выделяет в **динамической памяти**, а не на стеке.

std::vector имеет еще один важный атрибут: **емкость**. **Емкость вектора всегда больше или равна его размеру** и представляет количество элементов, которое вектор мог бы хранить в данный момент, прежде чем выделить дополнительную память для базового массива:

```
std::vector<char> v {42, 43, 44};
v.reserve(8);
```



векторы можно **копировать** (за **линейное время**) и сравнивать (`std::vector<T>::operator<` выполняет лексикографическое сравнение операндов, используя `T::operator<`).

vec.reserve(c) изменяет емкость вектора – она «**резервирует**» место в памяти для заданного количества `c` элементов. Если `c <= vec.capacity()`, тогда ничего не происходит ; но если

`c > vec.capacity()` , тогда происходит перераспределение памяти для внутреннего массива. (move для каждого элемента)

vec.resize(s) изменяет размер вектора – она усекает элементы в конце вектора (попутно вызывая их деструкторы) или добавляет в конец новые элементы (вызывая конструктор по умолчанию), пока размер вектора не станет равным `s`. Если `s > vec.capacity()` , происходит перераспределение внутреннего массива, как при вызове `.reserve()`.

Как определяется на сколько будет увеличиваться capacity?

capacityNew = capacity, умноженный на некоторый фактор роста (**growth factor**).

Фактор роста обычно больше 1.0, что позволяет амортизировать стоимость дополнительных операций выделения памяти при последовательном добавлении элементов. Наиболее распространенным является фактор роста **2 (gcc)**, хотя некоторые реализации могут использовать значения, например, **1.5** или другие, для достижения оптимального баланса между использованием памяти и производительностью.

Это означает, что если **capacity** вектора будет исчерпан и потребуются добавить еще один элемент, новый **capacity** будет рассчитываться по формуле, аналогичной следующей: **новый_capacity = старый_capacity * фактор_роста**. Точное значение фактора роста зависит от реализации стандартной библиотеки C++, которую вы используете.

QT отличие от QVector

Основные отличия

1. Принадлежность к библиотекам:

- **std::vector** является частью стандартной библиотеки C++ (STL).
- **QVector** является частью библиотеки Qt, фреймворка для разработки кроссплатформенных приложений.

2. Интеграция с библиотекой:

- `std::vector` лучше интегрируется с стандартной библиотекой C++ и алгоритмами STL.
- `QVector` лучше интегрируется с другими классами Qt, например, с `QString`, и поддерживает многие удобные функции Qt, такие как `foreach` и итераторы Qt.

3. Эффективность и производительность:

- `std::vector` обычно считается более низкоуровневым и, возможно, более эффективным в некоторых случаях, особенно когда речь идет о производительности и оптимизации памяти.
- `QVector` может предложить оптимизации, специфичные для Qt, такие как предварительное выделение памяти при добавлении элементов, что может улучшить производительность в некоторых Qt-специфичных сценариях.

4. API и функциональность:

- `std::vector` имеет API, ориентированный на стандартную библиотеку C++, что может делать его предпочтительным для общих задач программирования.
- `QVector` предлагает дополнительные функции, которые могут быть удобны при разработке приложений с использованием Qt, например, легкую интеграцию с другими классами Qt.

Как выбрать между `std::vector` и `QVector`?

- **Проект на Qt:** Если вы разрабатываете приложение с использованием Qt, `QVector` может быть более естественным выбором из-за лучшей интеграции и совместимости с Qt.
- **Требования к производительности:** Если производительность критически важна, может потребоваться провести бенчмарки, чтобы определить, какой из контейнеров лучше подходит для ваших конкретных требований.
- **Стандартные C++ проекты:** Для проектов, не использующих Qt, `std::vector` является стандартным выбором, так как он является частью стандартной библиотеки C++.
- **Кроссплатформенная разработка:** Если ваш проект должен быть кроссплатформенным и использует Qt для обеспечения совместимости, использование `QVector` может упростить разработку.

Выбор между `std::vector` и `QVector` в значительной степени зависит от контекста вашего проекта и от того, какие библиотеки и фреймворки вы используете. В большинстве случаев

Структура данных	Временная сложность								Сложность по памяти
	В среднем				В худшем				В худшем
	Индексация	Поиск	Вставка	Удаление	Индексация	Поиск	Вставка	Удаление	
Обычный массив	$O(1)$	$O(n)$	-	-	$O(1)$	$O(n)$	-	-	$O(n)$
Динамический массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Односвязный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Двусвязный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Список с пропусками	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Хеш таблица	-	$O(1)$	$O(1)$	$O(1)$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Бинарное дерево поиска	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Декартово дерево	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Б-дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Красно-черное дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Расширяющееся дерево	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
АВЛ-дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

4) Виды полиморфизма

Ответ:

В C++ полиморфизм — это концепция, позволяющая объектам принимать много форм. Полиморфизм в C++ преимущественно реализуется через:

- Статический (или компиляционный) полиморфизм:** Реализуется с помощью перегрузки функций и шаблонов.
 - Перегрузка функций** позволяет создавать несколько функций с одинаковым именем, но с различными параметрами. Компилятор определяет, какую функцию вызывать, исходя из списка аргументов при вызове.
 - Шаблоны** (особенно шаблоны функций и классов) позволяют функциям и классам работать с любым типом данных. Компилятор генерирует код для каждого конкретного типа, который используется с шаблоном, что позволяет достичь полиморфизма во время компиляции.
- Динамический (или выполнения) полиморфизм:** Реализуется с помощью виртуальных функций и наследования.
 - Виртуальные функции** позволяют классам-потомкам переопределять методы класса-родителя, объявленные как `virtual`. Во время выполнения программы используется механизм позднего связывания (или динамическое связывание), чтобы определить, какая версия функции должна быть вызвана, в зависимости от типа объекта, для которого она вызывается.

- **Чисто виртуальные функции и абстрактные классы** являются специальными случаями виртуальных функций. Чисто виртуальная функция не имеет реализации в базовом классе и обязывает производные классы предоставить свою реализацию. Класс, содержащий по крайней мере одну чисто виртуальную функцию, называется абстрактным классом.

Каждый из этих видов полиморфизма используется в зависимости от задачи:

статический полиморфизм используется для случаев, когда все данные о типе **известны на момент компиляции**, а **динамический** полиморфизм применяется, когда необходима гибкость и возможность определять **тип объекта во время выполнения**

5) Какими `_cast` пользовались?

Ответ:

C++ Casts

1. **`static_cast<>()`** - наиболее часто используемый способ приведения типов в C++. Он выполняет проверку типов на этапе компиляции и используется для преобразований между совместимыми типами, такими как преобразование базового класса в производный класс или наоборот, если между ними есть явная или неявная связь.
2. **`dynamic_cast<>()`** - используется для безопасного преобразования указателей и ссылок вниз по иерархии классов (от базового класса к производному). Он проверяет тип во время выполнения и возвращает нулевой указатель или выбрасывает исключение, если преобразование невозможно.
3. **`const_cast<>()`** - удаляет или добавляет квалификатор `const` к переменной. Это единственный способ изменить константность переменной без изменения её исходного определения.
4. **`reinterpret_cast<>()`** - используется для любых преобразований между типами указателей, включая преобразование между указателями и целыми числами. Это самый низкоуровневый и опасный cast, поскольку он позволяет преобразовывать любой указатель в любой другой тип указателя без проверки.

Qt-Specific Casts

В Qt5, помимо стандартных C++ приведений, используются также специализированные механизмы приведения типов, облегчающие работу с объектной моделью Qt:

1. **qobject_cast<>()** - аналог **dynamic_cast<>** для объектов, наследующих от **QObject**. Это безопасный способ приведения типов вниз по иерархии классов Qt. Если преобразование невозможно, возвращается нулевой указатель. Это приведение работает только с классами, объявленными с помощью макроса **Q_OBJECT**.
2. **Q_CAST** (устаревший) - макрос, который был использован в более ранних версиях Qt для приведения типов. В современных версиях Qt его использование не рекомендуется в пользу более типобезопасных методов, таких как **qobject_cast<>()**.

Важно отметить, что правильный выбор метода приведения типов зависит от конкретной задачи и требований к типобезопасности и производительности. В Qt **qobject_cast<>()** часто используется для безопасного приведения типов между объектами Qt, в то время как стандартные C++ приведения применяются в более общих ситуациях.

б) Что такое виртуальная таблица методов
Ответ:

Виртуальная таблица методов (часто называемая **vtable**) — это механизм, используемый в объектно-ориентированном программировании и, в частности, в C++, для поддержки динамического полиморфизма, то есть возможности вызывать методы объекта, тип которого не известен во время компиляции.

Каждый объект класса, содержащего хотя бы одну виртуальную функцию, имеет скрытый указатель на таблицу виртуальных методов этого класса. В этой таблице содержатся адреса всех виртуальных методов объекта. Когда вызывается виртуальная функция, программа использует указатель **vtable** объекта для определения того, какую реализацию метода следует вызвать. Это позволяет объектам разных классов реагировать на одни и те же сообщения (вызовы методов) разными способами, то есть реализуется полиморфное поведение.

Для чего нужна виртуальная таблица методов:

1. **Поддержка динамического полиморфизма:** **vtable** позволяет объектам разных типов обрабатывать одни и те же вызовы методов по-разному, основываясь на их фактическом типе, а не на типе, указанном при объявлении переменной. Это означает, что можно писать более общий и масштабируемый код.
2. **Позволяет реализацию позднего связывания (late binding):** решение о том, какой метод вызвать, принимается во время выполнения, а не во время компиляции. Это основной механизм для реализации полиморфизма во время выполнения.

3. **Упрощение кода при работе с иерархиями классов:** программисту не нужно заботиться о том, какой именно метод будет вызван. Система автоматически обеспечивает вызов корректной версии метода для объекта.

Например, представьте себе класс `Animal` с виртуальным методом `speak()` и два производных класса `Dog` и `Cat`, каждый из которых переопределяет метод `speak()`. Благодаря механизму vtable, при вызове `speak()` на объекте типа `Animal*`, который фактически указывает на объект `Dog` или `Cat`, будет вызван соответствующий метод `Dog::speak()` или `Cat::speak()`, даже если тип вызываемого объекта известен только во время выполнения.

- 7) Напишите макрос, который возводит число в квадрат

Ответ:

```
// очень важны скобки!!!
#define SQUARE(x) ((x) * (x))

#include <iostream>

int main() {
    int num = 5;
    int result = SQUARE(num);
    std::cout << "Square of " << num << " is: " << result <<
std::endl;
    return 0;
}
```

- 8) Для чего используется constexpr?

Ответ: ключевое слово `constexpr` в C++ используется для указания, что значение переменной или выражения может быть вычислено во время **компиляции**. Основная разница между `const` и `constexpr` заключается в моменте вычисления значения:

const: Переменная, объявленная с ключевым словом `const`, является константой, и ее значение известно только во время **выполнения программы**. Значение `const` переменной устанавливается во время компиляции и **не может быть изменено во время выполнения** программы.

constexpr: Переменная или функция, объявленная с ключевым словом `constexpr`, **может быть вычислена во время компиляции**, если все ее аргументы и операции также могут быть вычислены во время компиляции. Это позволяет выполнить оптимизации времени компиляции и повысить производительность программы. Значения `constexpr` могут использоваться в контекстах, где требует

Начнем с **ограничений** constexpr-переменных. Тип constexpr-переменной должен быть литеральным типом, то есть одним из следующих:

- Скалярный тип
- Указатель
- Массив скалярных типов
- Класс, который удовлетворяет следующим условиям:
 - Имеет деструктор по умолчанию
 - Все нестатические члены класса должны быть литеральными типами
 - Класс должен иметь хотя бы один constexpr-конструктор (но не конструктор копирования и перемещения) или не иметь конструкторов вовсе

constexpr-переменная должна удовлетворять следующим условиям:

- Ее тип должен быть литеральным
- Ей должно быть сразу присвоено значение или вызван constexpr-конструктор
- Параметры конструктора или присвоенное значение могут содержать только литералы или constexpr-переменные и constexpr-функции

Тут вроде ничего необычного. Основные ограничения наложены на constexpr-функции:

- Она не может быть виртуальной (virtual)
- Все параметры должны иметь литеральный тип
- Тело функции должно содержать только следующее:
 - `static_assert` (Проверяет программное утверждение во время компиляции. Если указано константное выражение `false`, компилятор отображает указанное сообщение, если он указан, и компиляция завершается ошибкой C2338; в противном случае объявление не действует.)
 - `typedef` или `using`, которые объявляют все типы, кроме классов и перечислений (enum)
 - `using` для указания видимости имен или пространств имен (namespace)
 - **Ровно** один `return`, который может содержать только литералы или constexpr-переменные и constexpr-функции

9) Какие бывают области видимости программы:

Ответ: глобальная область видимости: Это область, в которой имена доступны во всей программе. Переменные и функции, определенные в глобальной области видимости, могут быть использованы из любой части программы.

Локальная область видимости: Это область, ограниченная блоком кода, таким как функция, цикл или условие. Переменные и функции, объявленные внутри такой области, могут быть использованы только внутри этой области и не видны в других частях программы.

Область видимости блока: Это подтип локальной области видимости, который ограничивается конкретным блоком кода, выделенным фигурными скобками `{}`. Переменные, объявленные в таком блоке, могут быть использованы только внутри этого блока и не видны за его пределами.

10) Расскажите про принципы SOLID:

Ответ:

Принципы SOLID — это набор из пяти основных принципов объектно-ориентированного программирования и проектирования, которые помогают разработчикам создавать системы, которые легче поддерживать и расширять. Давайте рассмотрим, как эти принципы применимы к C++:

1. **S: Принцип единственной ответственности (Single Responsibility Principle - SRP):** Каждый класс должен иметь только одну причину для изменения. В контексте C++, это означает, что класс должен выполнять только одну задачу или иметь одну область ответственности. Например, класс для работы с файлами не должен заниматься непосредственно обработкой данных файла.
2. **O: Принцип открытости/закрытости (Open/Closed Principle - OCP):** Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации. В C++, это может быть реализовано через наследование и использование абстрактных базовых классов или интерфейсов, позволяя добавлять новые функциональности, не изменяя существующий код.
3. **L: Принцип подстановки Лисков (Liskov Substitution Principle - LSP):** Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы. Для C++ это означает, что наследующие классы должны полностью соответствовать интерфейсу и поведению базовых классов, так чтобы использование подкласса не нарушало ожидаемое поведение.
4. **I: Принцип разделения интерфейса (Interface Segregation Principle - ISP):** Клиенты не должны быть вынуждены реализовывать интерфейсы, которые

они не используют. В C++, это может быть достигнуто путем разделения больших интерфейсов на более мелкие и специфические, так чтобы классы могли реализовывать только те интерфейсы, которые им необходимы.

5. **D: Принцип инверсии зависимостей (Dependency Inversion Principle - DIP):** Модули высокого уровня не должны зависеть от модулей низкого уровня. Обе типа модулей должны зависеть от абстракций. Кроме того, абстракции не должны зависеть от деталей. В C++, это означает использование абстрактных классов или интерфейсов для определения зависимостей, вместо конкретных классов, позволяя таким образом снизить связность и увеличить гибкость кода.

Последнее означает, что нам **не нужно изменять существующую** кодовую базу для добавления новых функций, мы просто **строим поверх нее** или, что еще лучше, расширяем ее.

11) Расскажите про понятие RAII (понятие владения)

Ответ:

Понятие Владения

:: Объект А **владеет** объектом В означает, что объект А управляет жизненным циклом объекта В - то есть созданием, копированием, перемещением и уничтожением объекта В.

Пользователь объекта А не должен задумываться об **управлении В** (явный `delete В`, `fclose(В)` **правило 3**)

Деструктор Конструктор копирования Оператор присваивания копированием.

Важно понимать:

Хранение ≠ владение

хранение - часть от владения.

Понятие владения связано с RAII.

Более логичное название: Resource Free Is Destruction. **Освобождение ресурса - есть уничтожение.**

Этому принципу следуют и контейнеры. **Контейнеры владеют ресурсами, но итераторы - нет.** Пример: `remove_if` должен работать вместе с `erase`.

12) Что за служебное слово `explicit`

Ответ:

В C++, ключевое слово `explicit` используется с конструкторами для предотвращения неявных преобразований или копирований. В основном это делается для следующих целей:

1. **Предотвращение неявных преобразований типов:** Без `explicit`, конструктор, который принимает один аргумент, может быть использован для неявного преобразования типа. Это может привести к неожиданному поведению программы, когда компилятор автоматически преобразует типы данных. Использование `explicit` предотвращает такое преобразование, требуя от программиста выполнить преобразование типа явно.
2. **Предотвращение неявного создания временных объектов:** Когда функция принимает объект класса в качестве параметра, компилятор может использовать конструктор для создания временного объекта, если аргумент функции не соответствует типу параметра. Это может привести к дополнительным затратам на создание и уничтожение объекта. Ключевое слово `explicit` предотвращает создание таких временных объектов без явного указания.

Вот пример, иллюстрирующий различие между явными и неявными конструкторами:

```
class MyClass {
public:
    explicit MyClass(int value) {} // Требу
};

void function(MyClass obj) {}

int main() {
    // function(5); // Ошибка: нет подходящ
    function(MyClass(5)); // Работает, прео
    return 0;
}
```

Во втором случае, чтобы передать целочисленное значение функции, требуется явно создать объект `MyClass`, что уменьшает вероятность ошибок и улучшает читаемость кода.

13) Идиома `pimpl`

Идиома "pimpl" в C++ является сокращением от "Pointer to Implementation". Это паттерн проектирования, который используется для **уменьшения зависимостей компиляции путем уменьшения видимости деталей реализации класса**. Это достигается путем хранения **деталей реализации класса в отдельном классе**

реализации, на **который указывает указатель в определении интерфейсного класса**.

Основная идея заключается в том, чтобы определить в заголовочном файле интерфейсного класса только необходимый минимум информации, не раскрывая детали реализации, которые скрыты за указателем на реализацию. **Детали реализации класса описываются в файле исходного кода (.cpp), что позволяет изменять реализацию без необходимости перекомпиляции всех клиентских модулей, зависящих от этого класса**

QT

В Qt идиома **pImpl** часто используется и имеет специальное название – **"d-pointer"**. Qt применяет эту идиому во многих своих классах для обеспечения двоичной совместимости (binary compatibility) между разными версиями библиотеки. D-pointer является вариацией pImpl, где "d" обозначает "data" или "delegate", указывая на то, что этот указатель ведет к данным или делегирует выполнение внутреннему классу.

Основная цель использования d-pointer в Qt - **скрыть детали реализации класса от его интерфейса**, что позволяет разработчикам Qt изменять внутреннюю реализацию классов без необходимости перекомпиляции приложений, которые их используют. Это критически важно для обеспечения совместимости приложений с новыми версиями Qt без необходимости их изменения или перекомпиляции.

```
// header file
class MyClass {
public:
    MyClass();
    ~MyClass();

private:
    class Private;
    QScopedPointer<Private> d_ptr; // QScopedPointer
    обеспечивает автоматическое управление памятью
    Q_DECLARE_PRIVATE(MyClass) // Макрос для удобного
    доступа к d_ptr
    Q_DISABLE_COPY(MyClass) // Макрос для предотвращения
    копирования
};

// source file
class MyClass::Private {
public:
    int data; // Пример данных, скрытых от пользователя
    // ...
};

MyClass::MyClass() : d_ptr(new Private) {
}

MyClass::~~MyClass() {
    // QScopedPointer автоматически удаляет d_ptr
}

// Пример использования d_ptr для доступа к внутренним
// данным
void MyClass::someFunction() {
    Q_D(MyClass); // Получение доступа к d_ptr через макрос
    Q_D, определенный ранее
    d->data = 5; // Использование внутренних данных
}
```

В этом примере `QScopedPointer` используется для автоматического управления жизненным циклом объекта `Private`. Макросы `Q_DECLARE_PRIVATE` и `Q_D` предоставляют удобный доступ к внутреннему классу реализации. `Q_DISABLE_COPY` предотвращает копирование объекта, что является хорошей практикой при использовании идиомы `plmpl`.

Использование d-pointer идиомы в Qt обеспечивает эффективное разделение интерфейса и реализации, позволяет избежать лишних зависимостей в заголовочных файлах и облегчает поддержку двоичной совместимости между разными версиями Qt.

14) Правило трех (3) и правило пяти (5)

Правило трех в C++ гласит, что если классу нужен один из следующих трех специальных методов, то скорее всего ему понадобятся и два других:

1. Деструктор – для освобождения ресурсов, выделенных объектом.
2. Конструктор копирования – для создания нового объекта как копии существующего, с корректным копированием ресурсов.
3. Оператор копирования – для корректного копирования состояния одного объекта в другой существующий объект.

Правило пяти

С появлением стандарта C++11 в язык были добавлены перемещающие конструктор и перемещающий оператор присваивания, что привело к обновлению правила трех до "правила пяти". Правило пяти гласит, что если ваш класс реализует любой из следующих функций, то ему, вероятно, потребуется реализовать все пять:

1. Деструктор.
2. Конструктор копирования.
3. Оператор копирования.
4. Конструктор перемещения – для переноса ресурсов из одного объекта в другой, оставляя исходный объект в валидном, но неопределенном состоянии.
5. Оператор перемещением.

15) Что такое лямбда

Лямбда-выражения в C++ — это анонимные функции, которые можно использовать для создания компактных блоков кода, предназначенных для непосредственного исполнения. Лямбды были введены в стандарт C++11 и предоставляют мощный способ определения функций прямо в месте их использования, без необходимости именования и явного определения функций в другом месте.

Лямбда-выражения особенно полезны в следующих случаях:

1. **Краткосрочное использование:** Когда функция нужна для однократного использования, лямбда-выражения позволяют определить её прямо на месте, что упрощает чтение и поддержку кода.
2. **Функциональное программирование:** Лямбды облегчают использование парадигмы функционального программирования в C++, позволяя передавать функции как аргументы другим функциям, возвращать их из функций или присваивать переменным.
3. **Работа с коллекциями:** В сочетании с стандартной библиотекой шаблонов (STL), лямбда-выражения могут использоваться для эффективной обработки коллекций данных с помощью алгоритмов, таких как `std::sort`, `std::find_if`, `std::remove_if` и т.д.
4. **Замыкания:** Лямбда-выражения могут захватывать переменные из окружающего контекста, что позволяет им работать с данными, определёнными вне лямбды. Это делает лямбды мощным инструментом для создания замыканий.

Синтаксис лямбда-выражения в C++ выглядит следующим образом:

```
[захват](параметры) -> тип_возвращаемого_значения {  
    // тело функции  
}
```

- **захват** — определяет, какие переменные из внешнего контекста будут доступны внутри лямбды, и как они будут захвачены (по значению или по ссылке).
- **параметры** — список параметров, как в обычном определении функции.
- **тип_возвращаемого_значения** — необязательный параметр, определяет тип значения, возвращаемого лямбдой. Может быть опущен, если компилятор может определить тип автоматически.
- **тело функции** — код, который будет выполнен, когда лямбда вызывается.

16) `std::bind`

В C++, `std::bind` — это функция из стандартной библиотеки, которая позволяет создавать новые функции путем привязки аргументов к параметрам существующих функций. Это полезно в нескольких сценариях:

1. **Фиксация аргументов функции:** `std::bind` позволяет "заморозить" один или несколько аргументов функции, создавая новую функцию с меньшим числом аргументов. Например, если у вас есть функция, принимающая два аргумента, вы можете создать новую функцию, которая принимает только один аргумент, второй аргумент при этом будет фиксированным.
2. **Изменение порядка аргументов:** С `std::bind` можно изменить порядок аргументов функции при её вызове. Это может быть полезно, если вы хотите передать функцию в качестве коллбека, но порядок аргументов в функции-коллбеке отличается от ожидаемого.
3. **Использование в качестве адаптера для коллбеков и обработчиков событий:** `std::bind` часто используется для привязки объектов и их методов к интерфейсам, ожидающим функции определенной сигнатуры. Это делает `std::bind` мощным инструментом для работы с коллбеками и событиями в объектно-ориентированном коде.
4. **Совместимость с другими частями стандартной библиотеки:** Функции, созданные с помощью `std::bind`, можно использовать с алгоритмами стандартной библиотеки и другими компонентами, такими как потоки, асинхронные операции и таймеры.

Важно отметить, что с появлением лямбда-выражений в C++11, потребность в `std::bind` уменьшилась, так как лямбды предоставляют более гибкий и удобный способ достижения тех же целей. Лямбды позволяют инкапсулировать небольшие фрагменты кода в месте их использования, делая код более читабельным и лаконичным. Однако `std::bind` все еще может быть полезен в ситуациях, когда нужно использовать существующие функции или методы объектов без изменений.

17) Что такое perfect forwarding?

Perfect forwarding в C++ — это механизм, позволяющий функциям принимать аргументы любого типа (l-values или r-values) и передавать их другим функциям с сохранением исходного значения категории аргумента (l-value или r-value). Это достигается с помощью шаблонов функций и спецификатора `std::forward`.

Основная цель perfect forwarding — минимизировать количество лишних копий и операций перемещения при передаче аргументов между функциями, сохраняя при этом правильную семантику для l-values и r-values. Это особенно полезно **в шаблонах и обобщенном программировании**, где вы хотите, чтобы ваша функция могла работать с любыми типами аргументов.

Рассмотрим пример использования perfect forwarding:

```
template<typename T>
void wrapper(T&& arg) {
    // Передача arg дальше с сохранением категории l-value или r-value
    target(std::forward<T>(arg));
}

void target(int& x) { /* работа с l-value */ }
void target(int&& x) { /* работа с r-value */ }

int main() {
    int a = 5;
    wrapper(a); // a передается как l-value
    wrapper(5); // 5 передается как r-value
}
```

В этом примере `wrapper` использует perfect forwarding для передачи своего аргумента функции `target` с сохранением его исходной категории значения. Это достигается благодаря использованию `std::forward<T>`, которая "перенаправляет" аргумент в зависимости от того, был ли он передан как l-value или r-value. Таким образом, когда `wrapper(a)` вызывается с l-value аргументом, `target(int& x)` вызывается, а когда `wrapper(5)` вызывается с r-value аргументом, вызывается `target(int&& x)`.

В C++, l-value (locator value) и r-value (read value) являются двумя категориями выражений, основанными на их идентифицируемости и способности быть присвоенными.

L-value:

- **L-value** ссылается на объект, который занимает идентифицируемое местоположение в памяти (т.е., у него есть адрес). L-values могут появляться как слева, так и справа от оператора присваивания. Примеры l-value: переменные, элементы массива, индексированные значения, и дереференцированные указатели. Вы можете взять адрес l-value.

R-value:

- **R-value**, в контрасте, относится к временным и неидентифицируемым объектам, которые не занимают идентифицируемого местоположения в памяти. R-values могут появляться только справа от оператора присваивания. Примеры r-value включают литералы (например, `42`, `'a'`), временные значения (результаты выражений, таких как `2 + 2` или `std::move(x)`), и лямбда-выражения. R-values обычно используются для инициализации l-values или передачи данных функциям.

Расширение категорий:

С появлением C++11, категории значений были расширены и включают в себя:

- **xvalue** (expiring value): тип r-value, который представляет объект, готовый к "уничтожению", т.е., из которого можно извлечь ресурсы с помощью перемещения.
- **prvalue** (pure r-value): традиционное r-value, которое не связано с каким-либо объектом и часто используется для описания временных значений.
- **glvalue** (generalized l-value): обобщает l-values и xvalues, представляя выражения, связанные с объектом.

Раздел QT5

1) Отличие QThread от std::Thread

В QT реализуется механизм сигналов и слотов, поэтому QThread может обработать сигнал и по сигналу завершиться.




Если получатель находится в другом треде, то можно поставить тип соединения: Qt::QueuedConnection.

event handling (обработка событий) worker будет происходить внутри thread

QThread usage with an event loop (cont'd)

- The default implementation of `QThread::run()` actually calls `QThread::exec()`
- This allows us to run code in other threads without subclassing `QThread`:

```
1 auto thread = new QThread;
2
3 auto worker = new Worker;
4
5 connect(thread, &QThread::started, worker, &Worker::doWork);
6 connect(worker, &Worker::workDone, thread, &QThread::quit);
7
8 connect(thread, &QThread::finished, worker, &Worker::deleteLater);
9
10 worker->moveToThread(thread);
11 thread->start();
```






Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

For classes, the above definitions apply to non-static member functions when invoked on the same instance. (In other words, considering the `this` pointer as an argument.)



- **Thread safe:**
 - QMutex
 - QObject::connect()
 - QApplication::postEvent()
- **Reentrant:**
 - QString
 - QVector
 - QImage
 - value classes in general
- **Non-reentrant:**
 - QWidget (including all of its subclasses)
 - QQuickItem
 - QPixmap
 - in general, GUI classes are usable only from the main thread



QObject: queued connections (cont'd)

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
 - The event will then get dispatched to the object *by the right thread*
 - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
- Also, qRegisterMetaType() is required for the argument types passed
- We can force any connection to be queued:

```
connect(sender, &Sender::signal, receiver, &Receiver::slot, Qt::QueuedConnection);
```



```
auto initTcp = [=]() -> bool{
    if (!d->client) {
        d->client = new TcpClient();
        d->createConnections();
        // удаление объекта при disconnected сигнала
        QTcpSocket
        QObject::connect(d->client,
            &TcpClient::disconnectedFromServer,
            d->client, [=]() {
                d->client->deleteLater();
                d->client = nullptr;
            }, Qt::QueuedConnection);
    }
};
```

```

    }

    d->connectionStatus = d->connectToHost(d-
>waitMsec);
    return d->connectionStatus;
};

QMetaObject::invokeMethod(
    QAbstractEventDispatcher::instance(d-
>m_thread.get()),
    initTcp, Qt::BlockingQueuedConnection, &isOk);

```

2) Количество параметров у Qt::connect

Соединение объектов осуществляется при помощи статического метода *connect()*, который определен в классе *QObject*.

```

QObject::connect(
    const QObject*      sender,
                        const char*      signal,
    const QObject*      receiver,
    const char*         slot,
    Qt::ConnectionType type = Qt::AutoConnection
);

```

В общем виде *QObject::connect()* передаются пять следующих параметров.

- *sender* - указатель на объект, отправляющий сигнал.
- *signal* - сигнал, который отправляется объектом (на который указывает указатель *sender*).
- *receiver* - указатель на объект, принимающий сигнал.
- *slot* - слот объекта (на который указывает указатель *receiver*), вызываемый при получении сигнала.
- *type* - управляет режимом обработки. Имеет три возможных значения:
 - *Qt::DirectConnection* - сигнал обрабатывается сразу вызовом соответствующего метода слота.
 - *Qt::QueuedConnection* - сигнал преобразуется в событие и ставится в общую очередь для обработки.
 - *Qt::AutoConnection* - это автоматический режим, который действует следующим образом: если отсылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим *Qt::DirectConnection*, иначе - *Qt::QueuedConnection*.

Так же есть альтернативный способ задачи соединений через *QObject::connect()*, это:

```

QObject::connect(
    const QObject*      sender,

```

```

    const QMetaMethod& signal,
    const QObject*      receiver,
    const QMetaMethod& slot,
    Qt::ConnectionType type = Qt::AutoConnection
);

```

Параметры такие же, как и в приведенном выше, разница только в том, что в этом случае надо явно задавать имя класса, чей слот или сигнал вызывается, но при этом **ошибки** соединения сигналов-слотов **определяются на этапе компиляции**.

Можно задать через 3 параметра, если третий параметр this.

3) Model-view представление

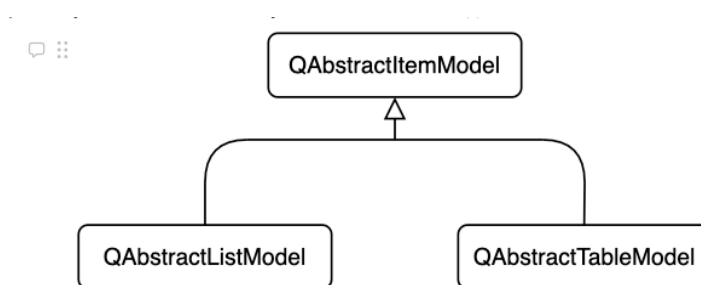
QComboBox -> модель + Виджет

Основная информация:

Модель - оболочка вокруг исходных данных, предоставляющая стандартный интерфейс для доступа к ним. Из-за того, что доступ к данным есть только через интерфейс, это значит, что модели можно разрабатывать отдельно друг от друга, потом лишь заменяя их.

В *Qt* все модели базируются на классе *QAbstractItemModel*.

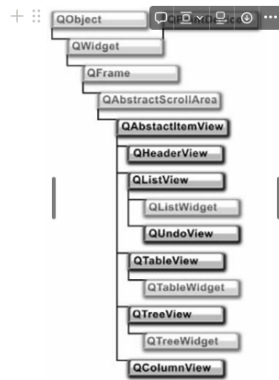
QAbstractItemModel - класс, определяющий стандартный интерфейс для Моделей элементов. Модель обязана использовать реализацию этого класса для корректного взаимодействия с другими элементами [архитектуры Модель/Представление](#). От этого класса не должно происходить явное создание экземпляров, приемлемо только наследование.



Класс *QAbstractItemModel* можно использовать для базового хранения данных для *Представлений* элементов в *QML* или же в *Виджетах Qt*.

Если стоит задача использовать в дальнейшем такие представления как *Представление Списка* в *QML*, либо же использовать виджеты [QListView](#) или [QTableView](#) то лучше наследоваться не от этого класса а от [QAbstractListModel](#) или [QAbstractTableModel](#), тк они реализует более частные случаи тех или иных моделей.

Представление



Иерархия наследования в
Представлении.

- *QAbstractScrollArea* - один из базовых классов для всех представлений, который позволяет использовать прокрутку по виджету.
- *QAbstractItemView* - так же один из важных базовых классов, дает всем наследникам

+ :: Делегат

Делегат - механизм представления, нужный для отрисовки элемента Модели и его редактирования пользователем. Другими словами: для стандартных Представлений списков и таблиц, отображение элементов выполняется по средствам так называемого делегирования.

Для создания какой-то сложной отрисовки или нетривиального редактирования, стоит создавать своих Делегатов при помощи наследования от класса *QAbstractItemDelegate*.



Иерархия наследования для Делегатов.

Раздел CMAKE

1) Что такое cmake и для чего нужен?

Ответ: cmake нужен для генерации make файла

2) Как указать альтернативный компилятор?

Ответ: объявив переменную среды `CMAKE_CXX_COMPILER` при C++ коде, или `CMAKE_C_COMPILER`, C код.

Пример: `set(CMAKE_CXX_COMPILER /путь/к/компилятору/g++)`

3) Какие есть варианты сборки в cmake? Как можно оптимизировать сборку проекта?

Ответ:

В CMake существует несколько различных вариантов сборки, которые могут быть указаны при генерации проекта. Некоторые из них включают:

- **Debug:** Для отладки приложения. Обычно включает отладочную информацию и отключает оптимизации компилятора.
- **Release:** Для выпуска конечной версии приложения. Обычно включает оптимизации компилятора и отключает отладочную информацию.
- **MinSizeRel:** Минимальный размер выпускаемого приложения. Этот вариант сборки использует максимальные оптимизации для минимизации размера исполняемого файла.
- **RelWithDebInfo:** Сборка для выпуска с отладочной информацией. Это сочетание оптимизаций для релиза с включенной отладочной информацией.

Вы можете указать вариант сборки при генерации проекта с помощью флага `-DCMAKE_BUILD_TYPE`, например:
bash

Пример:

`cmake -DCMAKE_BUILD_TYPE=Debug /path/to/source`

Это устанавливает тип сборки как Debug.

4) Пример cmake

```
cmake_minimum_required(VERSION 3.10.0 FATAL_ERROR)
project(StrataSolutionsWorkers)

set(CMAKE_INCLUDE_CURRENT_DIR ON) // включает автоматическое добавление текущих исходных
                                   // каталогов и каталогов сборки в путь include

set(CMAKE_AUTOMOC ON) // CMake будет автоматически обрабатывать препроцессор Qt moc,

set(CMAKE_AUTORCC ON) // автоматический запуск Resource Compiler с помощью rcc.

find_package(Qt5 REQUIRED COMPONENTS Core Gui Widgets Charts)

set(INCLUDES // tapret INCLUDES
    include/mainwindow.h
    include/generateWorker.h
)

set(SRC // tapret SRC
    src/main.cpp
    src/mainwindow.cpp
    src/generateWorker.cpp
)

set(FORMS // tapret FORMS
    forms/mainwindow.ui
)

set(RESOURCES // tapret RESOURCES
    resources/resources.qrc
)

qt5_wrap_ui(UI_HEADERS ${FORMS})

add_executable(
    ${PROJECT_NAME}
    ${INCLUDES}
    ${SRC}
    ${FORMS}
    ${RESOURCES}
)

// add_executable() указывает CMake создать исполняемый файл, можно указывать таргеты

target_include_directories(${PROJECT_NAME}
    PUBLIC
    ./include
    PRIVATE
    ./src
    ./forms
    ./resources
)

// target_include_directories делает: При сборке самой библиотеки, добавляет эту папку в список
// путей, в которых компилятор ищет инклюды. (Вероятно флагом -I, если это GCC.)
// То же самое делает при сборке того, что зависит от этой библиотеки.

Что говорить: target_include_directories добавляет путь к папке в список путей, которые
препроцессор будет использовать при поиске заголовочных файлов, включаемых в коде не в форме
относительного пути.

target_link_libraries(${PROJECT_NAME}
    PUBLIC
    Qt5::Core
    Qt5::Gui
    Qt5::Widgets
    Qt5::Charts
)

//target_link_libraries() указывает, что исполняемый файл зависит от библиотеки и должен быть
связан с ней при сборке.

target_compile_features(${PROJECT_NAME} PUBLIC cxx_std_17)
```

Совет: Используйте `target_compile_options` для добавления специфичных флагов компилятора, с которыми собирается цель.

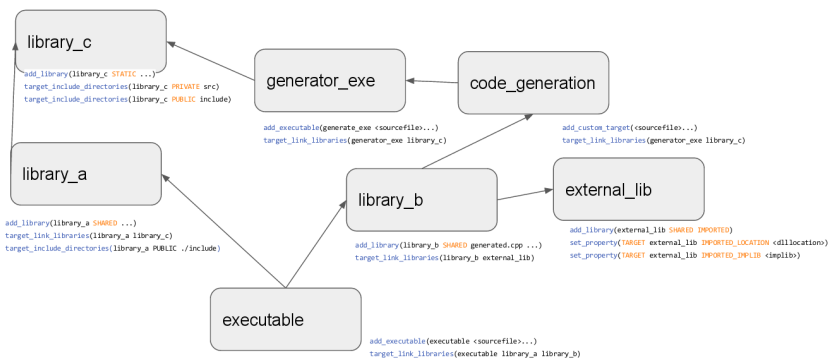
- **PUBLIC** означает, что зависящему от библиотеки проекту тоже нужны эти опции
- **PRIVATE** означает, что опции нужны лишь для сборки библиотеки
- **INTERFACE** означает, что опции не нужны для сборки, но нужны для использования библиотеки

```
# Способ первый: затребовать от компилятора фичу cxx_std_17
target_compile_features(${TARGET} PUBLIC cxx_std_17)

# Способ второй: указать компилятору на стандарт
set_target_properties(${TARGET} PROPERTIES
  CXX_STANDARD 17
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

Хорошая статья про современный cmake

<https://habr.com/ru/articles/330902/#>



5) Дополнительные флаги при сборке проекта? Чем они помогают?

Ответ: Могут ускорить сборку проекта.

<https://www.opennet.ru/docs/RUS/gcc/gcc1-2.html>

-o файл

Поместить вывод в файл 'файл'. Эта опция применяется вне зависимости от вида порождаемого файла, есть ли это выполнимый файл, объектный файл, ассемблерный файл или препроцессированный C код.

Поскольку указывается только один выходной файл, нет смысла использовать '-o' при компиляции более чем одного входного файла, если вы не порождаете на выходе выполнимый файл.

Если '-o' не указано, по умолчанию выполнимый файл помещается в 'a.out', объектный файл для 'исходный.суффикс' - в 'исходный.o', его ассемблерный код в 'исходный.s' и все препроцессированные C файлы - в стандартный вывод.

-v

Печатать (в стандартный вывод ошибок) команды выполняемые для запуска стадий компиляции. Также печатать номер версии управляющей программы компилятора, препроцессора и самого компилятора.

-pipe

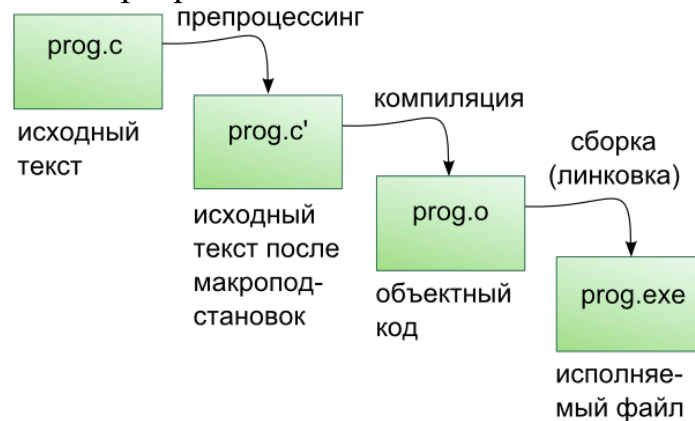
Использовать каналы вместо временных файлов для коммуникации между различными стадиями компиляции. Это может не работать на некоторых системах, где ассемблер не может читать из канала, но ассемблер GNU не имеет проблем.

Раздел СБОРКА ПРОЕКТА

1) Что такое DLL и виды библиотек?

Ответ:

Этапы компиляции программы



Ответ: **DLL (Dynamic Link Library)** - это формат библиотек в операционных системах Windows.

Например, user32.dll, kernel32.dll, gdi32.dll, sqlite3.dll которые содержат функции для работы с пользовательским интерфейсом, ядром операционной системы и графическими ресурсами и сетевыми протоколами.

Статическая библиотека - объектный модуль, код из которого выборочно или полностью вставляется в программу на этапе **компоновки** (линковки). При изменении чего-нибудь в библиотеке **требуется перекомпилировать** проект.

Объектный модуль - файл компиляции.

Транслятор переводит исходный код в машинный и записывает в объектный файл.

Модуль содержит:

- *машинный код;*
- *список своих процедур и данных;*
- *данные с неопределенными адресами ссылок на данные и процедуры в других объектных модулях.*

Соединяется с другими объектными файлами при помощи *компоновщика (linker)* (вычисляет и заполняет адреса перекрестных ссылок между модулями) в один исполняемый файл.

Статические библиотеки:

- Статическая библиотека компилируется непосредственно в исполняемый файл программы во время компиляции. (boost)
- Все функции и данные из статической библиотеки копируются в исполняемый файл программы.
- Это означает, что исполняемый файл самодостаточен и не требует наличия внешних файлов для работы.
- Однако это может привести к увеличению размера исполняемого файла и дублированию кода при каждой компиляции.

Динамическая библиотека - файлы подпрограмм на машинном коде, которые загружаются в приложение во время **выполнения**. Когда вы компилируете программу, использующую динамическую библиотеку, библиотека не становится частью вашего исполняемого файла – она остается отдельной единицей.

Плюсы:

- экономия памяти за счёт использования одной библиотеки несколькими процессами;
- возможность исправления ошибок (достаточно заменить файл библиотеки и перезапустить работающие программы) без изменения кода основной программы.

Недостатки:

- возможность нарушения API — при внесении изменений в библиотеку существующие программы могут перестать работать (утратят совместимость по интерфейсу);
- конфликт версий динамических библиотек, — разные программы могут нуждаться в разных версиях библиотеки;
- доступность одинаковых функций по одинаковым адресам в разных процессах — упрощает эксплуатацию уязвимостей (для решения проблемы изобретён `pic` [статья на эту тему](#)).

Одна из ключевых идей, на которых основывается PIC, – смещение между секциями `text` и `data`, размер которого известен линкеру во время линковки. Когда линкер объединяет несколько объектных файлов, он собирает их секции вместе (к примеру, все секции `text` объединяются в одну большую секцию `text`). Таким образом, линкеру известны и размеры секций, и их относительное расположение.

Разделяемая библиотека - подтип *динамической библиотеки*, которая содержит функции используемые несколькими программами. Могут загружаться в адресное пространство ОС для экономии памяти: одна копия библиотеки будет использоваться несколькими процессами.

```
#if defined(DLL_LIBRARY)
#define DLL_EXPORT Q_DECL_EXPORT
#else
#define DLL_EXPORT Q_DECL_IMPORT
#endif
```

Где Q_DECL_EXPORT и Q_DECL_IMPORT на препроцессинге подставляются как:

```
# define Q_DECL_EXPORT __declspec(dllexport)
# define Q_DECL_IMPORT __declspec(dllimport)
```

Из мануала по Qt:

Depending on your target platform, Qt provides special macros that contain the necessary definitions:

- Q_DECL_EXPORT must be added to the declarations of symbols used when compiling a shared library;
- Q_DECL_IMPORT must be added to the declarations of symbols used when compiling a client that uses the shared library.

Макрос Q_DECL_EXPORT в QT5 используется для объявления символов из динамически подключаемой библиотеки (DLL) **при компиляции** динамической библиотеки.

Макрос Q_DECL_IMPORT в QT5 используется для объявления **экспорта символов из динамически подключаемой библиотеки** (DLL), когда они используются в других модулях приложения.

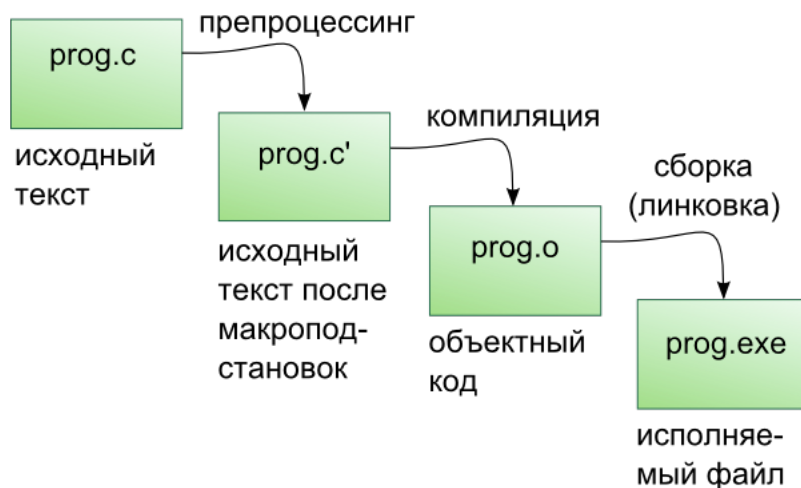
Динамические библиотеки:

- Динамическая библиотека загружается и связывается с программой во время выполнения.
- В исполняемом файле программы сохраняется лишь **ссылка на динамическую библиотеку**, а не сам код и данные из нее.
- Это позволяет сократить размер исполняемого файла и обеспечить возможность обновления или замены библиотек без перекompиляции программы.

<https://github.com/Danykrane/Qt-flow/blob/master/2%20less/readme.md>

2) Какие этапы происходят при компиляции программы?

Ответ:



Препроцессор — это *макро процессор*, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит работа с препроцессорными директивами. Например, препроцессор добавляет хэдеры в код (**#include**), убирает комментирования, заменяет макросы (**#define**) их значениями, выбирает нужные куски кода в соответствии с условиями **#if**, **#ifdef** и **#ifndef** (**Include Guard**).

Однако, каждый хедер может быть открыт во время **препроцессинга** **несколько раз**, поэтому, обычно, используются специальные препроцессорные директивы. `ifdef endif pragma`

Pragma vs ifdef endif

Фундаментальное отличие заключается в том, что `#pragma once` относится ко всему заголовочному файлу целиком, а `ifdef endif` можем ограничить область включения для единицы трансляции.

Соответственно, ответ очевиден - пользуйтесь стандартной функциональностью `ifdef/endif` и забудьте про нестандартный `#pragma once`. Может быть в каком-то случае вы заметите, что компилятор (препроцессор) не хочет оптимизировать обработку на основе анализа `ifdef/endif` и использование `#pragma once` действительно ускоряет компиляцию... Тогда, если для вас это критично - может быть стоит добавить в ваши файлы `#pragma once`.

Получим **препроцессированный** код в выходной файл **driver.ii** (прошедшие через стадию препроцессинга C++ файлы имеют расширение **.ii**), используя флаг **-E** ([флаги тут](#)) который сообщает компилятору, что компилировать (об этом далее) файл не нужно, а только провести его препроцессинг:

```
g++ -E driver.cpp -o driver.ii
```

Компиляция:

- трансляция (программа после препроцессинга в ассемблерный код)

- ассемблирование (ассемблерный код в объектный (машинный))

```
$ g++ -S driver.ii -o driver.s  
$ as driver.s -o driver.o
```

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о *таблице символов*.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.

Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

- Таблица символов – массив структур
- Каждая запись содержит имя, размер, позицию символа.

```
g++ driver.o -o driver // также тут можно добавить и другие объектные файлы и библиотеки
```

Если внутри единицы трансляции (один файл после препроцессинга) у объекта или функции есть внутреннее связывание то этот символ виден компоновщику только внутри этой единицы трансляции. Если же у объекта или функции есть внешнее связывание, то компоновщик сможет видеть его при обработке других единиц трансляции.

Использование ключевого слова **static** в глобальном пространстве имен дает символу внутреннее связывание. Ключевое слово **extern** дает внешнее связывание.

Компилятор по умолчанию дает символам следующие связывания:

- Non-const глобальные переменные — внешнее связывание;
- Const глобальные переменные — внутреннее связывание;
- Функции — внешнее связывание.

3) Какие бывают виды связывания? Ограничение видимости для единицы трансляции.

Ответ:

Внешнее связывание имеют те сущности, к которым можно обратиться в единице трансляции, отличной от той, где они определены.

```
extern int object;
```

Внутреннее связывание отличается от внешнего тем, что к сущности имеющей внутреннее связывание нельзя обратиться из единицы трансляции, отличной от той, где она определена. Так, если взять тот же пример, но изменить first.cpp на

```
static int object;
```

или на

```
namespace  
{  
    int object  
}
```

4) Как можно ускорить сборку?

Ответ: флагами компиляции

Для **gcc**:

```
gcc -O2 -S source.c
```

Для **Clang**:

Copy code

```
clang -O2 -S -emit-llvm source.c
```


Раздел Lixux

1) Как вывести список процессов

Ответ:

- `ps` - список процессов текущего пользователя;
- `ps e` - список процессов на системе;

```
artemgudzenko@192 ~ % ps
  PID TTY          TIME CMD
 57592 ttys000    0:00.05 -zsh
artemgudzenko@192 ~ % ps e
  PID  TT  STAT      TIME COMMAND
 57592 s000  S        0:00.05 -zsh
```

- `ps aux` - подробный список **всех** процессов на системе;

```
artengudzenko@192 ~ % ps aux
USER      PID %CPU %MEM    VSZ   RSS Tt  STAT  STARTED    TIME COMMAND
_         1      0  0  0  0  0  Ss   29янв24 116:37.01 /usr/sbin/coreau
root      191    0  0  0  0  0  Ss   29янв24 99:36.26 /usr/libexec/air
artengudzenko 11408  0,6  2,8 498760400 441456 ?? Ss   13фев24 14:10.44 /System/Library/
artengudzenko 32822  0,5  1,8 414641280 272556 ?? S   1:21    5:14.66 /Applications/Mi
_windowsserver 176    0  4  1 41304864 178018 ?? S   29янв24 199:40.53 /System/Library/
artengudzenko 1231    0  3  0 2 414298096 40144 ?? Ss   30янв24 22:17.00 /System/Library/
root      189    0  2  0 2 408406176 39568 ?? Ss   29янв24 6:42.06 /usr/libexec/log
root      1    0  2  0 2 409482144 26624 ?? Ss   29янв24 20:11.18 /sbin/launchd
artengudzenko 57286  0,1  0,4 409831888 65328 ?? S   10:36    0:01.15 /System/Applicat
artengudzenko 11502  0,1  3,1 500175840 524720 ?? Ss   13фев24 7:52.42 /System/Library/
root      172    0  1  0 0 408265968 6000 ?? S   29янв24 4:30.52 /usr/libexec/oc
artengudzenko 472    0  1  0 0 408170992 5632 ?? S   29янв24 1:43.03 /usr/sbin/distno
artengudzenko 42608  0,1  2,8 516258096 46688 ?? Ss   6:38    1:39.55 /System/Library/
artengudzenko 48386  0,0  0,0 408714176 5888 ?? S<   8:09    0:00.02 /Applications/Vi
artengudzenko 48380  0,0  0,1 408967984 9120 ?? Ss   8:09    0:00.09 /System/Library/
artengudzenko 48390  0,0  0,1 40897352 22768 ?? S   8:09    0:00.05 /Applications/Vi
artengudzenko 48179  0,0  0,0 409118816 5648 ?? Ss   8:09    0:00.07 /System/Library/
artengudzenko 48178  0,0  0,2 441874160 31056 ?? S   8:09    0:01.95 /Applications/Vi
artengudzenko 48177  0,0  0,4 441948080 70240 ?? S   8:09    2:14.00 /Applications/Vi
artengudzenko 48176  0,0  0,0 441684928 5264 ?? S   8:09    0:00.04 /Applications/Vi
artengudzenko 48170  0,0  0,0 1585813344 150880 ?? S   8:09    0:41.73 /Applications/Vi
artengudzenko 48166  0,0  0,0 408102944 3344 ?? Ss   8:09    0:00.02 /System/Library/
artengudzenko 48165  0,0  0,1 408285712 21184 ?? Ss   8:09    0:00.12 /System/Library/
artengudzenko 44687  0,0  0,2 513957088 28408 ?? Ss   7:10    0:00.11 /System/Library/
artengudzenko 43808  0,0  0,2 408297808 30304 ?? S   6:57    0:03.38 /System/Library/
artengudzenko 43782  0,0  0,3 408288736 42496 ?? S   6:57    0:01.95 /System/Library/
artengudzenko 43777  0,0  0,1 408392352 22768 ?? S   6:57    0:00.31 /System/Library/
artengudzenko 42740  0,0  0,0 408968512 3616 ?? Ss   6:40    0:00.04 /System/Library/
artengudzenko 42743  0,0  0,0 408958272 3644 ?? Ss   6:40    0:00.04 /System/Library/
artengudzenko 42740  0,0  0,1 409116992 10032 ?? Ss   6:40    0:00.37 /System/Library/
artengudzenko 41847  0,0  0,0 408958272 3664 ?? Ss   6:24    0:00.04 /System/Library/
artengudzenko 41846  0,0  0,0 409248064 3792 ?? Ss   6:24    0:01.20 /System/Library/
artengudzenko 41819  0,0  1,8 515402624 162112 ?? Ss   6:24    0:22.64 /System/Library/
artengudzenko 41817  0,0  0,0 408958272 3644 ?? Ss   6:24    0:00.03 /System/Library/
artengudzenko 41816  0,0  0,0 409107776 3792 ?? Ss   6:24    0:00.30 /System/Library/
artengudzenko 41768  0,0  0,5 499003760 77840 ?? Ss   6:23    0:25.97 /System/Library/
artengudzenko 41439  0,0  0,0 408958272 3616 ?? Ss   6:18    0:00.04 /System/Library/
artengudzenko 41440  0,0  0,0 408958272 3744 ?? Ss   6:18    0:00.12 /System/Library/
artengudzenko 40304  0,0  0,0 408228016 3712 ?? S   5:10    0:01.45 /Library/Apple/S
root      40119  0,0  1,1 408386992 8448 ?? Ss   5:17    0:02.04 /Library/Apple/S
artengudzenko 30550  0,0  1,1 408386992 8448 ?? Ss   5:05    0:00.35 /Library/Apple/S
```

- **top** - список процессов с сортировкой по критериям;

```
Processes: 656 total, 2 running, 654 sleeping, 2466 threads
Load Avg: 1.15, 1.14, 1.26 CPU usage: 5.28% user, 4.70% sys, 90.1% idle
SharedLibs: 614M resident, 112M data, 40M linkedit.
MemRegions: 13777 total, 5421M resident, 372M private, 1756M shared,
PhysMem: 15C used (1502M wired, 2833M compressor), 805M unused,
VM: 255T vszize, 4272M framework vszize, 272449(0) swaptins, 328064(0) swaptouts,
Networks: packets: 11685020/9767M in, 11194400/3999M out. Disks: 6233876/151G read, 3678924/97G written.

PID COMMAND %CPU TIME #TH #WQ #PORT MEM PURG CMPRS PGRP PPID STATE BOOSTS
176 WindowServer 31.2 03:20:16 21 6 2755 630M+ 33M+ 141M 176 1 sleeping *0[1]
0 Kernel task 9.1 02:45:30 492/0 0 0 2394K+ 0 0 0 0 0 running *0[0]
57987 screencaptur 0.0 00:00:03 2 1 66 8018K+ 752K 0 512 512 sleeping *0[847+]
57954 top 6.5 00:02:44 1/1 0 31 8225K 0 0 57954 57592 running *0[1]
385 com.apple.Ap 4.0 01:44:12 4 3 132 3137K 0 1184K 385 1 sleeping *0[1]
186 coreaudiod 3.5 01:56:47 8 1 1270 50M 0 50M 186 1 sleeping *0[1]
12296 bluetoothd 2.4 15:43:11 11 5 308+ 11M+ 272K 12M 12296 1 sleeping *0[1]
23328 Telegram 1.8 11:38:61 35 6 533 696M 768K 384M 23328 1 sleeping *0[2233]
57968 screencaptur 1.1 00:00:38 3 1 198 11M 0 0 57968 1 sleeping *0[492+]
12408 com.apple.We 0.0 14:12:40 5 1 106 264M 0 44M 11408 1 sleeping *0[72229]
48177 Code Helper 0.0 02:14:49 21 3 152 67M 4544K 11M 48170 48170 sleeping *1[5]
1231 com.apple.We 0.5 22:19:19 20 1 502 74M 128K 55M 1231 1 sleeping *0[118682]
1 launchd 0.4 20:15:28 5 3 3699 31M 0 0 5264K 1 0 sleeping *0[0]
582 chardm 0.4 07:18:49 6 3 3624 21M+ 0 7684K 582 1 sleeping *0[1]
11582 com.apple.We 0.3 07:53:67 5 2 99 614M 0 141M 11582 1 sleeping *0[995199+]
57286 Terminal 0.3 00:04:48 6 1 256 30M 3408K 0 57286 1 sleeping *0[82]
42688 com.apple.We 0.3 01:40:78 4 1 105 441M 8416K 81M 42688 1 sleeping *0[174828+]
189 top 0.3 05:42:08 5 4 2153 13M 0 0 9728K 189 1 sleeping *0[1]
472 distnoted 0.2 01:43:22 2 1 504 4017K 0 592K 472 1 sleeping *0[1]
546 useractivity 0.2 00:15:02 4 3 109+ 4465K+ 0 720K 546 1 sleeping *0[4798]
192 runningboard 0.2 07:17:31 7 6 705+ 9297K 0 592K 192 1 sleeping *1[3414]
662 nearby 0.2 01:05:35 7 5 94+ 4689K 0 2432K 662 1 sleeping *1[10833]
1229 com.apple.We 0.2 13:35:92 18 3 253 208M 960K 101M 1229 1 sleeping *0[22486]
122 powerd 0.1 02:53:40 3 2 137 4817K 0 976K 122 1 sleeping *0[1]
168 distnoted 0.1 00:59:45 2 1 238 285K 0 912K 168 1 sleeping *0[1]
167 opendirector 0.1 03:19:77 8 7 1752 12M 128K 2352K 147 1 sleeping *0[1]
378 audiodlocksy 0.1 02:47:13 6 5 79 5025K 0 2448K 378 1 sleeping *0[1]
191 airtportd 0.1 09:37:53 9 7 6574 40M 0 20M 191 1 sleeping *9[108159]
15329 com.apple.We 0.1 00:36:17 15 1 118 147M 0 194M 15329 1 sleeping *0[10611]
178 notifyd 0.1 02:41:47 3 2 921 3825K 0 848K 178 1 sleeping *0[1]
526 identityserv 0.1 00:52:55 8 2 358+ 16M+ 288K 4192K 526 1 sleeping *0[1]
172 corebrightness 0.0 04:30:74 2 1 133 4369K 0 1840K 172 1 sleeping *0[1]
32836 com.apple.m 0.0 00:35:38 3 1 79 3921K 0 1864K 32836 1 sleeping *0[6115]
511 rapportd 0.0 01:48:24 2 1 210 7825K 0 1952K 511 1 sleeping *0[1]
247 mDNSResponder 0.0 04:49:15 3 1 97 7329K 0 960K 247 1 sleeping *0[1]
19831 relative 0.0 00:15:66 4 2 83 3921K 0 2512K 19831 1 sleeping *0[74543+]
308 cald 0.0 00:13:00 4 3 32 3649K 0 1232K 308 1 sleeping *0[1]
522 WirelessRadi 0.0 00:52:84 2 1 63 3329K 0 1376K 522 1 sleeping *0[1]
1926 storagekid 0.0 00:34:53 9 5 67 31M 0 29M 1926 1 sleeping *0[49]
```

- Более удобную статистику можно посмотреть с помощью **htop**

Для скачивания нужно прописать в консоли:

Mac Os: **brew install htop**

Linux: **sudo apt-get install htop**

PID	USER	PRI	NI	VIRT	RES	S	CPU%	MEM%	TIME+	Command
11985	artemgudze	17	0	391G	138M	?	50.6	0.8	10:19.00	/System/Applications/Notes.app/Contents/MacOS/Notes
59943	artemgudze	17	0	398G	20896	?	6.2	0.1	0:00.00	/usr/sbin/screencapture -pdi -z keyboard.selection
23328	artemgudze	17	0	393G	508M	?	1.3	3.1	11:40.00	/Applications/Telegram.app/Contents/MacOS/Telegram -noup
57288	artemgudze	24	0	391G	102M	?	1.1	0.6	0:11.00	/System/Applications/Utilities/Terminal.app/Contents/Mac
59944	artemgudze	40	0	390G	24080	?	0.9	0.1	0:00.00	/System/Library/CoreServices/screencaptureui.app/Conte
11408	artemgudze	17	0	475G	421M	?	0.8	2.6	14:16.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
42608	artemgudze	17	0	492G	456M	?	0.7	2.8	1:44.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
1229	artemgudze	17	0	395G	110M	?	0.7	0.7	13:40.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
48177	artemgudze	17	0	421G	70176	?	0.6	0.4	2:15.00	/Applications/Visual Studio Code.app/Contents/Frameworks
11502	artemgudze	17	0	477G	515M	?	0.6	3.1	7:55.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
1231	artemgudze	17	0	395G	40800	?	0.5	0.2	22:21.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
59902	artemgudze	24	0	390G	8560	?	0.5	0.1	0:00.00	htop
59917	artemgudze	17	0	389G	20384	?	0.3	0.1	0:00.00	/System/Library/Frameworks/CoreServices.framework/Framework
509	artemgudze	17	0	389G	16336	?	0.0	0.2	0:11.00	/usr/libexec/nsurlsessiond
540	artemgudze	8	0	389G	39408	?	0.1	0.2	7:30.00	/System/Library/PrivateFrameworks/CloudKitDaemon.framework
15329	artemgudze	17	0	491G	65952	?	0.1	0.4	0:36.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
48178	artemgudze	17	0	421G	31024	?	0.0	0.2	0:02.00	/Applications/Visual Studio Code.app/Contents/Frameworks
582	artemgudze	24	0	389G	31328	?	0.0	0.2	7:19.00	/usr/libexec/sharingd
517	artemgudze	17	0	389G	4432	?	0.0	0.0	0:26.00	/System/Library/CoreServices/lockoutagent
58104	artemgudze	17	0	490G	217M	?	0.0	1.3	0:04.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
56690	artemgudze	24	0	1511G	63504	?	0.0	0.4	0:00.00	/Applications/Visual Studio Code.app/Contents/Frameworks
570	artemgudze	17	0	389G	14528	?	0.0	0.1	0:52.00	/System/Library/PrivateFrameworks/HearingCore.framework/
32822	artemgudze	17	0	395G	297M	?	0.0	1.8	5:37.00	/Applications/Microsoft Word.app/Contents/MacOS/Microsof
1225	artemgudze	8	0	396G	399M	?	0.0	2.4	15:45.00	/System/Volumes/Preboot/Cryptexes/App/System/Application
665	artemgudze	17	0	389G	2608	?	0.0	0.0	0:05.00	/usr/libexec/gamecontrolleragentd
16319	artemgudze	17	0	475G	141M	?	0.0	0.9	1:03.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
513	artemgudze	17	0	391G	108M	?	0.0	0.7	1:24.00	/System/Library/CoreServices/Finder.app/Contents/MacOS/F
12101	artemgudze	17	0	492G	86656	?	0.0	0.5	1:52.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
808	artemgudze	17	0	389G	24480	?	0.0	0.1	4:39.00	/usr/libexec/searchpartyuseragent
14459	artemgudze	17	0	490G	146M	?	0.0	0.9	0:21.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X
795	artemgudze	17	0	389G	784	?	0.0	0.0	0:00.00	/System/Library/PrivateFrameworks/Categories.framework/V
41768	artemgudze	17	0	475G	77984	?	0.0	0.5	0:26.00	/System/Library/Frameworks/WebKit.framework/Versions/A/X

2) Как удалить процесс?

Ответ: `kill -9 PID_NAME`

Выводим список всех процессов и смотрим их PID:

```
1 | ps aux
```

Фильтруем по имени процесса, например apache2:

```
1 | ps aux | grep apache2 (grep – поиск строки)
```

Можно также посмотреть PID процесса выполнив команду:

```
1 | pidof apache2
```

Список сигналов и их номеров которые можно послать процессу:

```
1 | kill -l
```

Убиваем процесс (где PID — номер процесса, стандартно процессу шлется сигнал SIGTERM):

```
1 | kill PID
```

Убиваем процесс с указанием сигнала (например 9):

```
1 | kill -9 PID
```

Пример завершения процессов по имени:

```
1 killall apache2  
2 killall -s 9 apache2
```

Справки по командам:

```
1 man ps  
2 man grep  
3 man pidof  
4 man kill  
5 man killall
```