**Слайд 1**

Презентація орієнтована на Андроїд девелоперів в яких вже є досвід роботи з платформою андроїд і які тільки починають інтегрувати С++.

Або С++ девів які починають працювати з Андроїд платформою.

В цілому в презентації буде йти мова про типові помилки з якими можуть стикатись інженери і короткі поради як уникати таких проблем.

Я не буду розповідати про те як настроювати JNI підходи і як з ним працювати а буду фокусуюсь на тих проблемах з якими ми дуже часто зтикаємося на нашому проекті. Як приклад

**Слайд 2**

Я хочу сконцентрувати увагу на таких проблемах як типові помилки з роботою з ЖНІ посиланнями, проблеми які можуть виникнути при роботі з нативними потоками їхній взаємодії, також пару порад стосовно виключень.

Поговорим як зменшити розмір кінцевих апк файлів і як це потім впливає на роботу креш стеками

**Слайд 3**

Що таке НДК -

необхідний набір інструментарію для розробки компонентів програмного забезпечення для платформи [Android](https://uk.wikipedia.org/wiki/Android), який базується на [C](https://uk.wikipedia.org/wiki/C)/[C++](https://uk.wikipedia.org/wiki/C%2B%2B) та інших мовах програмування. Містить в собі лімітований набір загальновживаних низькорівневих (нативних) бібліотек та [API](https://uk.wikipedia.org/wiki/API), написаних на С/С++ та інших мовах програмування, документацію і мінімальний набір прикладів для демонстрації базового функціоналу. За допомогою NDK розробник застосунку для операційної системи Android може імплементувати окремі його частини, використовуючи такі мови, як C/C++

ЖНІ -

стандартний механізм для запуску коду, під управлінням [віртуальної машини Java](https://uk.wikipedia.org/wiki/%D0%92%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_Java) (JVM), який написаний на мовах [С](https://uk.wikipedia.org/wiki/C_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F))/[С++](https://uk.wikipedia.org/wiki/%D0%A1%2B%2B) чи Ассемблер, та скомпонований у вигляді динамічних бібліотек, дозволяє не використовувати статичне зв'язування. Це дає можливість викликати функції С/С++ з программи на [Java](https://uk.wikipedia.org/wiki/Java), і навпаки

**Слайд 4**

Для чого взагалі використовувати NDK якщо це приводить до збільшення часу написання коду і також на відладку?

Перш за все дуже багато старих проектів які були написані на С++ а потім потрібно інтегрувати під мобільну платформу. В нашому випадку андроїд. Скільки різних ігор написаних на ОпенГЛ було імпортовано на андроїд.

Інше це різні приклади роботи з секюріті і перфоманс. Більшість андроїд бібліотек які пропнують подібна жава інтервейси насправді делегують виклики до системних бібліотек.

Кожен може розпакувати апк файл і побачити хоч і обфускований але код. ДЕХ то ДЖАР і тому поідбні утиліти допомагають в цьому.

Для чого ми використовуємо НДК на проекті?

Але все таки на мою думку найбільша перевага це все таки можливість написання коду який одночасно можна викорустувати на різних платформах.

Наприклад н анашомк проекті ми пишемо СДК яке розповсюджується на та різні мобільні платформи та також для макос і віндовс десктопів.

Структура проекту. Ми підтримуємо 5 різних платформ – перелічити. І для того щоб не імплементувати для кожної платформи наше СДК АПІ, в нас є С/С++ ядро. Воно покриває

Різні секьюрті протоколиб шифрування і збереження даних , нетворкінгб ауненифікація,

Різні внутрішні протоколи і тому подібне.

І коли для обжектівСІ та Віндовс Win32 нема проблем напряму комунікувати з С++ кодом (не беручи до уваги ), то для джави нам потрібно додавати додатковий рівень \_ ЖНІ.

Що може додами кучу проблем. З прешої точки зору кодм може виглядати цілком нормально, але різні під час роботи програми ми можемо зтикатися з різними проблема.

Якщо ми детектимо проблему під час дебагу то це цілком нормально. Але, якщо вже в релізному продукті – то це нетавні відгуки кастомерів. Тож полговоримо про типові помилки -

**Слайд 5**

**Слайд 6**

Перше на що я хочу звернути увагу – це локальні та глобальні посилання.

Для нормального збереження та передачі даних між С++ та Джавою ми використовуємо структру даних. В принципі це фраппер над виділенним ресурсом. Локально буде автоматчно очистена ДЖВМом як тільки ми покинемо нативний метод. І тому якщо ми хочемо зберегти її на деякий час потрібно зробити з неї глобальне посилання. Глобальна в свою чергу не буде очищена поки ми не почистимо її вручну.

Але:

Хоч і система чистить локальні посилання автоматично, все таки це не так. Всі локалні посилланя будуть очищені як тільки курування покине нативний метод і повернеться до віртуальної машини, і тому локалні посилання будуть накопичуватись (і навіть перезапис не допоможе). Що в результаті може приветси до такого крушу(показати креш)

Як бачимо по дефолту система дозволяє нам дозволяє створити лише 512 локальних посилань – це і є таблиця локильних посилань. Ми навіть можемо побачити декілька останніх в креш репорті. І тому тут можуть опинитися навіть паролі.

Ми можемо збільшити розмір даної таблиці але найкращий варінт одразу оищати посилання в ручну як тільки воно стає не поібним. (Приклад коду).

Це саме стосцється і глобальних посилань, але правда розмір таблиці більший(код і креш) але все одно не потрібно зловживати ними. Як порада – мінімізувати використання глобальних посилань.

Також якщо зберігаєте джава пір обекти як глобальні посилання в С++ коді – думайте що обьетк вже можу бути не валідним на стороні Джави. Наприклад актівіті –

Код.

В андроїді 8+, системо підтримує безлімітну кілкість локальних посилань, але все одно – не забувайти релізити- кількість 8 андроїдів хоч і зростає але все одно більшість девайсів працють на старіших версіях.

**Слайд 7**

При роботі з ЖНІ рекомендується зменшувати взаємодію з декільками потоками і по можливості робити все в одному потоці, але з розростанням проекту ми

можемо створювати велику кількість нативних потоків безпосередню в С++(POSIX, C++11) які можуть взаємодіяти між собою, але іноді виникає необхідність викликати якісь специфічні функції на стороні платформи. Як приклад – стан нетворку, телефонії, оріентація екрану, бекграунд тощо.

З кожною версією НДК набір АПІ збільшується, але все одно така необхідність буде залишатися.

Якщо ми просто викличемо якийсь Жава метод через JNI API в нативному потоці, то просто отримаємо креш – чому?

JNI визначає дві основні структури даних - "JavaVM" і "JNIEnv". Обидва ці по суті є вказівниками на покажчики на функціональні таблиці.

JNIEnv забезпечує більшість функцій JNI. Але JNIEnv використовується для доступу виключно до threаd local storage. З цієї причини ви не можете поділитися JNIEnv між потоками.

Для цього JavaVM надає функції "інтерфейсу виклику.

AttachCurrentThread або AttachCurrentThreadAsDaemon

До тих пір, поки потік не буде додано потоку, він не має власного JNIEnv і не може робити виклики JNI. Приєднання потоку до ЖВМ створить об'єкт java.lang.Thread на стороні жави, який буде доданий до "main" ThreadGroup, роблячи його видимим для дебагера.

Такі потоки, додані через JNI, повинні викликати DetachCurrentThread, перш ніж вони виходять.

[JNI\_OnLoad](https://docs.oracle.com/javase/9/docs/specs/jni/invocation.html) функція використовується для отримання поінтера на ЖВМ.

Для цього використовуємо функцію [GetEnv](https://docs.oracle.com/javase/9/docs/specs/jni/invocation.html).

Також це саме стосується різних локальних посиланью. Їх не можн ашарити між потоками. Тобто

Навітіь якщло нативний метод одного потоку ще не закінчив роботу а ми створили локальне посилання і додали його в якийсь глобальний контейнер, інший потік не може його використовувати.

Код. Шляхи вирішення

**Слайд 8**

Тепер пару слів про типові проблеми лінковики Джава і С++.

Перше часто під час написання коду і подальшому запуску ми не бачимо ніяких проблем під час компіляції а вже потім в рантаймі ми отримуємо креші такого типу (перелічити).

Типові помилки: змінили сігнатуру методу в джаві, перевірили Андроїд студією що всі викліки коректні. Але потім JNI вже в рантаймі не може знайти імплементацію

Чи змінили сігнатуру функцій ЖНІ хедеру.

Також не потрібно забувати що ЖНІ функції мають С а не С++ тип лінковки, і тому не потрібно забувати додавати екстерн С . В С++ ми можемо перегружати функції і методи і тому для правильної ліоквки С++ компілятор додає додаткову інформацію про функції в її ІД.

Дуже часто на нашому проекті ми зтикаємо с з такою – пишемо новий функціонвл на стороні Джави і С++ компілимо тестимо на локальній машині – все працює – сабмітимо зміни в репозиторій. Але через деякий хтось каже що ми поламали білд – в чому проблема –

обфускації жава коду в релізнму білді яку ми не перевірили.

Після обфускації жава код стає не хюман редабле що також є проблемою для ЖНІ де ми також покладаємося на точні імена класів і методів.

Тут як варінт нам потрібно додавати різні виключення в правила настройки прогварда і дексгварда.

Також якщо додаємо пітримку різних архтектур як (перелічити) – потрібно пересвідчитися що і інші бібліотеки які ми додамо в проект також пітримують дані архітектури. Приклад

Ми релізимо сдк з пітримкою 3 архітектур. 64 платформа якщо не знайде найбіль підходящу арх то вибере наприклад 32. Якщо вибере

Варто зазначити всі перераховані проблеми на етапі компіляції не детектаться а можуть лише в рантаймі і навіть тільки після якихосб специфічних дій користувача і тому тут потрібно приділяти дуже багато уваги до таких проблем.

**Слайд 9**

Інше на чому хочу звернути увагу – це підтримка експшинів в С++, JNI та джаві –

ексепшни в С++ ця функціональність є не опціональною і можуе бути вимкнена для підвищення швидкодії і зменшення розміру бібліотк і виконуваних програм.

І тому якщо ми під час компіляції бачимо щось подібне – то значить фічу потрібно вімкнути.

Для цього додаємо це як С++ білд флаг. .

Тут одне вмикає інше вимикає вічц відповідно..

Під час таких змін варто робити клін білду і рефрещити прилінколваний С++ проект (В вндроїд студії)

**Слайд 10**

Далі – наприклад ми хочемо викликати якись метод на стороні Джави який може генерувати експпш. Дуже багато Джава АПІ які кидають різні ксепшини які не є фатальними для роботи програми. І якщо експшн був кинутий на стороні Джави то на повернуний результат ми не можемо покладатися хоча ми і отрумумо результат В с++. Якщо в потоці висить пендінг ексепшин то ми більше не можемо викликати інші методі на стороні жави через JNI API. Є лімітований набір методів які ми можеми викликати поки у нас висить пендінг ексепшн здебільшого звязаний з очисткою ресурсів. Якщо ми спробуємо викликати інші методи – JNI викине помилку і за абортить програму.

Для цього нам після виклики потрібно перевіряти пендінг експшнини. Для цього викоритосвуємо дану функцію –

Для того зоб отримати тип експшина ми може використати функцію. Вона повертає jthrowable який ми може розпарсити і отримати тип ексепшина. Але Find class не може бути викликаний поки э пендінг – або чистимо експшн або кешуємо класс на ще до виклику методу. Ось типовий приклад - код

Ексепшн можна почисти функцією і працювати далі – в іншому випадку отримаю креш на рівні апплікейшина.

Також для Для підтримки стилю джави ми можемо кидати експшн з С++ коду на сторону Джави.

Для цього використовуємо пару функцій –

Приклад ось

Хочу зазначити що на стороні джави ескпшн буде викинутий тільки коли нативний метод поверне керування в ЖВМ. Тож навіть після того як ми викликали ThrowNEw ми можемо ще виконати якысь дії на стороні С/С++

**Слайд 11**

Додавання С++ рівня в проект додає суттево в розмірі остаточного АПК файлу і тому одна з задач яке перед нами може стати це зменшення бібліотек.

Перше що тут скащати що на выдміну від джава коду який компілюється в dex формат без різниці яка архітеркура процесора – це задача АРТ чи далвіка для більш старих платформ –які вже потім будуть самі адаптувати код під платформу. – то з С біблотеками нам потрібно предаставляти по лібі для кожної платформи – ось типова релізний апк файл – бачимо що ми маємо один декс файл і аж чотири динамічні со бібліотеки. Тут потрібно думати чи хочемо ми супортити ту чи іншу платформу і чи потрібно наприклад супортити 64 арм архітектури бо іноді може бути достаньо і 32 – вони компатібл але не навпаки . Інтел мало розповсюджені але емулятори саме інтел.

Також як було зазначено, використання експшинів приводить до генерації додаткового коду і що значить до збільшення фінальної бібліотеки –використанням флагів компілтора можна вмикати вимикати фічу.

RTTI – також фіча мови С++ що дозволяє в рантаймі дізнаватися тип обьекта і тому піодбне – це також приводить до генерації додаткового коду компіляторм. ТАкож можна вимкнути якщо це вам не потрібно.

Більше користі нам дасть зрізання символів і таблиць символів які можуть йти разом з динамісною бібліотекою.

TODO

**Слайд 12**

Для додавання всіх параметрів компіляціх нам потрібно передати ці флаги компілятору.

Зараз Android рекомендує використовувати смаке як тула для білда ндк коду, хоча ndk\_ build ще також підтримується. Він використовує лайт версію маке файлів – applicaiotn mk

Якщо використовуємо Смаке то можна передавати чи через білд градл – наприклад

Чи напряму в Смаке ліст ткст файл – варто зазначити що нам окремо потрібно передавати флаги для с++ так і для С.

Також можна вибирати релізний чи дебажний режим..

В аplication mk файлі це буде мати вигляд – . Окремо можна також передавати флаги лінковки – на цьому уваги акцентувати не буду.

**Слайд 13 Crash handling**

Так всі маніпуляції з кодом було виконано – все оптимізовано все працює – ми релізимось в плей маркет – і отрмуємо перші репорти в Dev consoli –

**Слайд 14**

Приклад типового крешу який можна знайти на гугл дев консолі в Android Vitals .

Як бачимо – типова статисткиа по версім продукту, андроїда і пристроям.

І беспосередньо сам креш –

Як бачимо не дуже багато корисної інформації для початку. Бектрейс не показує ніяким методів чи символів (на відміну від джава стеку)

Зменшення розміру Зрізання різних символів приводить до таких наслідків як повністю обфускований стек крешу. Це корисно для різного захисту від пен тестерів але як бачимо ми також не можемо отримати миттєво ніякої інформаціїї.

Тож давайте поглянемо на більш детальний tombstone креш і розберемося що для чого

І як та інформація що в нас є допоможе нам

**Слайд 15**

When a dynamically linked executable starts, several signal handlers are registered that, in the event of a crash, cause a basic crash dump to be written to logcat and a more detailed "tombstone" file to be written to /data/tombstones/. The tombstone is a file with extra data about the crashed process. In particular, it contains stack traces for all the threads in the crashing process (not just the thread that caught the signal), a full memory map, and a list of all open file descriptors.

Приклад детального крешу згенерованого на тестовому проекті –

The line of asterisks with spaces is helpful if you're searching a log for native crashes. The string "\*\*\* \*\*\*" rarely shows up in logs other than at the beginning of a native crash.

1 – The fingerprint lets you identify exactly which build the crash occurred on

The revision refers to the hardware rather than the software. This is usually unused but can be useful to help you automatically ignore bugs known to be caused by bad hardware.

The ABI is one of arm, arm64, mips, mips64, x86, or x86-64. This is mostly useful for the stack script mentioned above, so that it knows what toolchain to use.

This line identifies the specific thread in the process that crashed. In this case, it was the process' main thread, so the process ID and thread ID match. The first name is the thread name, and the name surrounded by >>> and <<< is the process name. For an app, the process name is typically the fully-qualified package name (such as com.facebook.katana), which is useful when filing bugs or trying to find the app in Google Play. The pid and tid can also be useful in finding the relevant log lines preceding the crash.

This line tells you which signal (SIGABRT) was received, and more about how it was received (SI\_TKILL). The signals reported by debuggerd are SIGABRT, SIGBUS, SIGFPE, SIGILL, SIGSEGV, and SIGTRAP. The signal-specific codes vary based on the specific signal.

2 - регістри

The register dump shows the content of the CPU registers at the time the signal was received. (This section varies wildly between ABIs.) How useful these are will depend on the exact crash.

3 - bactrace

The backtrace shows you where in the code we were at the time of crash. The first column is the frame .The PC values are relative to the location of the shared library rather than absolute addresses. The next column is the name of the mapped region (which is usually a shared library or executable, Finally, if symbols are available, the symbol that the PC value corresponds to is shown, along with the offset into that symbol in bytes.

The tombstone contains the same information as the crash dump, plus a few extras. For example, it includes backtraces for *all* threads

**Слайд 16**

Що нас тут цікавить так це сам бектрейс бо тільки він присутній в типових репортах на Гугл дев консолі.

Для декодування і отримання більш детальної інформації по бексрейсу нам потрібна така утіліта як ндк стек.

Інструмент ndk-stack дозволяє фільтрувати обфусковані стеки, як вони з'являються на виході adb logcat. Він також замінює будь-яку адресу в бібліотеці за допомогою відповідних значень <source-file>: <number-line> з вашого вихідного коду, що полегшує їх виявлення.

На вході вона нам потрібно передати песпосередньо сам бектрейс і шлях до бібліотеки з включенеми таблицею символів - вона може бути знайдена в білд оутпуті cімека чи ндк білда (приклад смаке) і тому варто завжди зберігати такі дані десь на сервері і мапати їх до тої версії апплікейшинів.

Також, потрібно вказувати версію білотеки для тої архітектури де стався креш.

Прогнати команду – показати результат

**Слайд 17**

**Показати приклад що було що стало**

**Слайд 18**

Ми також можемо написати свій креш хендлер. Для цього його потрібно правильно ініціалузіувати

**Слайд 19**

Але під час обробки сигналів ми не можемо використовувати майже нічого через

Більшість інструментів