> По компилятору:  
> 1) Какие именно расширения языка отключает «-pedantic-errors»?

У опций -std=имя стандарта для каждого языка C/C++ и года есть два варианта, официальный (c/c++) и с GNU-шными расширениями (gnu/gnu++). Второй вариант принимает все виды GNU-шных расширений, в том числе, когда они меняют смысл программ, строго соответстующих стандарту. Первый - только в том случае, где они не противоречат ему (например, конструкции, которых в стандарте нет вообще). Т.е. даже с первой опцией некоторые не входящие в язык конструкции принимаются. Чтобы случайно не влезть в расширения (например, массивы нулевой длины), опция -pedantic-errors превращает использование любых расширений сверх стандарта в ошибки. Подробнее о нюансах в man gcc, поиск по слову Wpedantic.

> 2) Какие ещё warning добавляет «-Wall»?

"Основные", что под этим подразумевается зависит от компилятора и версии.  
Для clang'a зависимость разных опций ворнингов описана в документации:  
https://clang.llvm.org/docs/DiagnosticsReference.html  
Для gcc то же есть в man gcc, поищите все вхождения Wall.

> 3) Какие отличия std=gnu++17 от std=c++17? (хочется знать, что мы отключаем)

См. ответ на вопрос 1. Если конкретно, то вот списки расширений (для gcc, clang старается поддерживать основные из них):  
https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html  
https://gcc.gnu.org/onlinedocs/gcc/C\_002b\_002b-Extensions.html  
Кроме некоторых мелочей ничего особо нужного там нет, львиная доля полезного там делает вещи из новых стандартов доступными в старых.

> 4) Какая отладочная информация сохраняется в Debug сборке?

Это очень общий вопрос с множеством деталей для конкретной реализации. Таблицы соответствия строк исходного текста программы и машинного кода, чтобы можно было проходить по строкам, расположение автоматических объектов на стеке функций, чтобы показать при останове значения локальных переменных и многое-многое другое. Повспоминайте разные возможности отладчика, в большинстве своём всё из этого вытекает.

> По CMake:  
> 1) Где у QtCreator задание стандартного CMake-файла?

Разный для разных типов проекта, посмотрите в /usr/share/qtcreator/templates/wizards/projects .  
Нарочно не стал вам в виртуалке прописывать сразу нужное, потому что всё равно нужно вас приучить всё прописывать самостоятельно для других сред.

> 2) В чём отличие системы сборки от семантики сборки?

В том, что второго термина не существует. Если у меня такое проскочило, я имел под этим в виду подмножество того, что необходимо указать в CMakeLists.txt для корректной сборки, в противовес необязательным вещам (ворнинги, санитайзеры, оптимизации и прочее). Мы обсуждали, что жёсткое прописывание такие вещей в CMakeLists.txt мешает тем, кому это не нужно. ntc-cmake разрабатывался с учётом того, чтобы автоматически прописывать кучу таких вещей для удобства, но иметь возможность всё выключить одной опцией.

> 3) Почему используется ninja вместо make?

Быстрее работает за счёт того, что разрабатывался именно как второе звено после, скажем CMake или Meson, а не полноценная система сборки с человекочитаемым файлом конфигурации с богатыми возможностями.

> 4) Как указать количество работников Jobs генератору через cmake? (чтобы не писать вручную ninja -j 4)

Это make'у надо -j указывать руками для параллельной сборки. Ниндзя по умолчания использует все ядра.

> 5) Можете привести примеры, когда нужно указывать target\_sources?

Вы имеете несколько платформозависимых реализаций чего-либо, которые слишком длинные, чтобы переключаться  условной трансляцией в одном файле исходного текста. Тогда размещаем их в отдельных. При сборке на конкретной платформе понадобится только один из них. В параметрах add\_executable/library идут исходники, которые собираются всегда, а вот дополнительные для конкретной плафтормы можно добавить через target\_sources.

> 6) Можете привести пример использования target\_compile\_options с проверкой на доступность?

Фактически, в ntc-cmake. Там модуль ntc-dev-build собирает в переменную NTC\_COMPILE\_OPTIONS всё, что было распознано компилятором, а в ntc-target-helpers это задаётся. Или можно так, отдельно:

# Попробовать сделать тип char беззнаковым.  
include(CheckCXXCompilerFlag)  
check\_cxx\_compiler\_flag(-funsigned-char HAVE\_UNSIGNED\_CHAR)  
if(HAVE\_UNSIGNED\_CHAR)  
    target\_compile\_options(${PROJECT\_NAME} PRIVATE -funsigned-char)  
endif()

> 7) В лекциях сказано «INTERFACE-библиотеки являются контейнерами INTERFACE\_\* свойств».   
> Для чего могут использоваться эти свойства и можете ли привести примеры, на которые стоит обратить внимание?

В первую очередь, через INTERFACE\_INCLUDE\_DIRECTORIES приходят каталоги include'ов, так что такие цели рассматриваются в первую очередь как header-only библиотеки. Для примера с другим свойством, например, можно посмотреть модуль FindThreads:

# Выяснить, как включить поддержку многопоточности на данном компиляторе  
# современным образом:  
set(THREADS\_PREFER\_PTHREAD\_FLAG ON)  
find\_package(Threads REQUIRED)  
# Интерфейсная библиотека Threads::Threads, найденная выше, содержит необходимые для  
# работы потоков характеристики. Для Visual C++, где всё всегда и так включено,  
# на ней ничего не задано. Для линукса обычно это INTERFACE\_COMPILE\_OPTIONS=-pthread,  
# для других плафторм это могут быть и флаги компилятора, и компоновщика, и макросы  
# предопределённые, и каталоги включаемых файлов...  
target\_link\_libraries(${PROJECT\_NAME} PRIVATE Threads::Threads)

> 8) Не очень ясна разница между add\_library и find\_package.  
> Правильно ли понимать, что find\_package ищет проект cmake и подключает его? Зачем тогда нужен add\_library (является ли он аналагом add\_executable для указания цели)?

add\_library/add\_executable создают цель в этом проекте, которая будет им собираться (если не IMPORTED/INTERFACE) и состоит из исходников в этом проекте. Обычное их использование - без IMPORTED для целей самого проекта.  
find\_package ищет файлы конфигурации CMake, которым передаёт управление, чтобы они создали через add\_library IMPORTED цели с указанием всех необходимых для использования параметров, соответствующих библиотекам, уже собранным (не важно, как) и установленным в системе. Как зависимости для target\_link\_libraries потом можно указывать цели и оттуда и оттуда, при правильном подходе, см. ntc-cmake, можно сделать даже универсальный вариант.

> 9) В документации упоминается ещё один тип библиотек cmake “OBJECT-библиотеки”. Для чего они нужны?

Файлы, указанные исходниками для таких библиотек, компилируются, но никуда не компонуются. Список полученных объектных файлов можно извлечь выражением из свойств цели (см. документацию) и использовать в других целях. Иногда используется в хитрых сценариях сборки. Например, если вас не смущает, что в статической библиотеке будет позиционно независимый код, можно собрать группу единиц трансляции один раз, а потом дважды собрать из этого и статическую и динамическую библиотеку без двойной компиляции.

> 10) Какой стандартный CMAKE\_MODULE\_PATH в windows? Как его настроить?

Если вы поставили cmake в windows, он, как и в любой другой ОС указывает на идущие с ним в комплекте модули. Можно вывести это значение или посмотреть, что за файлы установлены. Также, в любом случае указание дополнительных путей префиксов через CMAKE\_PREFIX\_PATH расширяет одновременно и путь модулей CMake соответствующими путями из префикса. Ну а вручную вы его обычно расширяете путями, относительными к вашему проекту.

> 11) Где хранятся файлы Pkg-config в windows?

Как и в случае с CMake, в той иерархии каталогов, которая соответствует его установке. В windows он, насколько я знаю, просто так не ставится, только как часть сред эмуляции UNIX, соответственно искать в их файлах.

> 12) Считается ли модуль cmake установленным, если файл «somemodule.cmake» был скачан на компьютер и перемещён в папку CMAKE\_MODULE\_PATH? Найдёт ли его find\_package?

В документации find\_package точно описаны все каталоги в системе и префиксах, где ищутся файлы, и какие имена. Имена должны быть вида ИмяПакетаConfig.cmake или имя-пакета-config.cmake. Список каталогов большой, рассчитанный на разные ОС и среды, см. докуемнтацию.  
Чтобы пакет считался "найденным" помимо успешной загрузки его конфигурации ещё нужно, чтобы файл версий загрузился и ответил утвердительно на вопрос соответствия имеющейся версии затребованной.

> 13) Зачем interface библиотеке так же указывать и IMPORTED? (пример из лекций: add\_library(SparsePP::SparsePP INTERFACE IMPORTED) )

IMPORTED означает, что эта цель - описание использования уже готовой цели, подготовленной вне этого проекта. STATIC/SHARED/INTERFACE выбирает тип библиотеки: статическая (собирать архиватором)/динамическая (собирать компоновщиком)/заголовочная (собирать не надо). Ваш пример из модуля поиска  header-only библиотеки, поэтому она и INTERFACE IMPORTED - "внешняя, заголовочная". У не-INTERFACE IMPORTED цели необходимо задать свойство IMPORTED\_LOCATION с указанием файла самой библиотеки, а в этом случае его нет.

> По c++:  
> 1) Помнится, что на лекции по теме «владение» были хорошие примеры для понимания этого понятия. Остались ли они у вас?

Должны быть в любой версии книжки, или там не всё?

> 2) В материалах к лекциям есть упоминание «Uuidguard». Кроме того, не раз замечал у вас в хедерах конструкцию вида UU\_ID\_\*. Не могли бы вы напомнить, что это такое и как это получить?

UUID - Universally Uunique IDentifier - 128-битные значения, способы генерации их в автономном режиме, минимизирующие коллизии с другими такими попытками и формат записи в виде текста как сгруппированные значения шестнадцатеричных цифр с тире. Имеют множество применений, в данном случае речь идёт о генерации уникальных идентификаторов header guard'ов, не опирающихся на имена самих файлов. uuidguard - это мой скрипт, который есть в виртуалке для генерации шаблонов заголовков такого вида. Он просто использует утилиту командой строки uuidgen из пакета util-linux:

$ cat `which uuidguard`  
#!/bin/sh  
uuid="UUID\_$(uuidgen | tr a-z- A-Z\_)"  
echo -e "#ifndef ${uuid}\n#define ${uuid}\n\n#endif"

> 3) До сих пор не смог понять отличия std::forward от std::move. Я понимаю, что std::move меняет семантику объекта на временную, но вот что делает std::forward и в каких случаях?

std::move(x) - это static\_cast к праводопустимой ссылке для того, что является изменяемым, т.е. придание изменяемому значению категории xvalue и, следовательно rvalue, если это возможно. Это делает его более подходящим для соответствующих перегрузок, отвечающих за перемещение. Вы его используете, чтобы сказать о значении "мне оно больше не нужно, можно украсть".

forward<T>(x) используется при прозрачной передаче аргументов сквозь вашу функцию. Для этого соответствующий параметр делается типа T&&, где T - типовой параметр шаблона, параметр такого типа называется универсальной/форвардящей ссылкой, для которой дедукция, подстановка и привязка проходят для любого типа, изменяемости и категории значения. Внутри функции с таким параметром имя соответствующего параметра имеет те же характеристики, что и аргумент, за исключением того, что категория значения - всегда lvalue, т.к. это имя ссылки. Прежде чем указывать это значение для передачи дальше, его надо прогнать через forward<T>, который вернёт его к rvalue, если по T видно, что оно таким изначально было. Таким образом, forward используется для передачи значений с сохранением всех свойств - прозрачной передачи аргументов.

Путаница ещё из-за того, что если их раскрыть руками, оба выглядят как static\_cast<T&&>(x) - только move к этому приходит дедукцией (аргумент шаблона не нужен), а forward требует явного указания соответстующего параметра шаблона. Это совпадение, по сигнатурам с метафункциями видно, что роль у них принципиально разная.

4) В чём отличие private от protected поля? Может ли это был полезно когда-нибудь, кроме абстрактных классов?

Не только поле, но и любое описание с уровнем доступа protected доступно не только самому классу, но и унаследованным (напрямую или косвенно) от него. Если говорить именно о protected членах данных, то это члены данных, с которыми напрямую могут взаимодействовать наследники. Действительно, встречается мнение, что поля делать protected плохо, т.к. инвариант работы с полем должен быть инкапсулирован в одном классе, без открытия дыры на произвольное количество потомков, которые инварианты могут нарушить, в таком стиле protected бывают только функции с полезной для производных классов функиональностью, а все поля private (не рассматривая случай с public полями класса без инкапсуляции вообще). Я лично привык потомкам доверять и перевожу поля из private в protected в момент, когда надо с ними сделать что-то из производного класса, что иначе потребовало бы создания акцессоров только ради этого. Иначе придётся делать сами акцессоры protected, но тогда уж надо формально рассматривать два вида "интерфейса класса" - для внешнего использования, и для использования наследниками. Кто-то любит это оформлять формально. Я - нет. Вопрос скорее стилевой в плане педантичности следования понятию инкапсуляция, ИМХО.