**a) What are the main motivations for adopting modern C++ features, such as lambda expressions, range-for, and declarations using the auto operator?**

Participant 1

My personal motivations for adopting new C++ features has been to write code which is easier to understand and more concise, as well as the convenience that these new features bring.

To use the examples that you stated: Lambda expressions allow the skipping of boiler-plate code when the code is small enough to fit within the context where it is used, thus eliminating the need to jump to a different code section while reading the code.

Range-for offers (when it can be used) a far superior syntax compared to classic iterator based code. In fact, Qt had introduced Q\_FOREACH before range-for was available because iterator-based loops were clearly a pain point before. Transitioning from that Qt-specific solution to the generic range-for was a no-brainer...

There is a slightly finer line with auto - I never was a "almost always auto" guy. My (again, personal) reason for adopting auto is readability and maintainability of code. Therefore, I use auto when it allows me to write more concise code and to avoid repeating myself. And sometimes it's simply convenient to let the compiler infer the type name for me.

Participant 2

The main reason is to make code nicer and simplify where it's possible. I would be very careful about adopting some C++ features because it's easy to abuse them, e.g. the auto keyword.

Participant 3

Well… it depends on the feature.  
  
My main motivation is safety. Qt historically had a huge weakness in compile  
time safety regarding its signal/slot system. It has been solved in Qt 5, but  
it relies on C++11 features (as far as I remember) that were thus unavailable  
before.  
ie. before you wrote:  
connect(obj1, SIGNAL(something()), obj2, SLOT(anotherthing()));  
translated by the preprocessor into:  
connect(obj1, "something()", obj2, "anotherthing()");  
and processed at run time. A huge source of bugs, making refactoring harder…  
Now you write:  
connect(obj1, &Class1::something, obj2, &Class2::anotherthing);  
  
Lambda expressions allow a much more readable code when writing callbacks or  
connecting to the signal of a short-lived object. Instead of having functions  
spread in a file, or worse having to write silly classes just to keep the  
context of a callback, now we have everything generated by the compiler.  
Same goes, readability wise, for auto.  
Range-for was already kind of present through some Qt macro kung-fu, so it's  
more getting rid of this magic to switch to the proper way of doing things,  
and getting some performance improvements by removing useless copy  
constructions…

Participant 4

In my opinion it is to have easier to read and simpler code. E.g. if you have some long type (especially with C++ template types it might be harder to read than just auto).  
range-for idea is not that different from Qt's Q\_FOREACH which was already used a lot, so it makes sense to move to C++ standard version (and range-for generates  
slightly better machine code than Q\_FOREACH). Lambda expressions are again useful to write simpler code, especially with Qt signals and slots.

Participant 5

My main motivation is better legibility and more concise and expressive  
code. I believe all three lead to code that is easier to maintain and less  
error-prone.  
  
I also believe that using modern C++ features makes KDE software more  
attractive for new contributors.

Participant 6

In the case of lambda expressions: They allow to write logic in places  
were it contextually fits, removing one layer of indirection when trying  
to understand the code. They are also considerably faster to write.  
  
With range-for, code becomes easier to read, by allowing to express the  
intention instead of indirectly archiving the same semantic by the use  
of indices or iterators. In KDE, the difference is not so large though,  
since we had the Qt foreach / Q\_FOREACH macro before which was similar.  
  
With auto, especially when using templated classes, writing new code is  
much more comfortable, since long type signatures don't need to be  
spelled out every time. Also, when changing the type of a variable, with  
auto the amount of code that needs to be changed becomes a lot less.  
However, it might even sacrifice some of the readability to speed of  
development. At least while using IDEs, this isn't a problem in practice.  
  
Generally speaking, mostly if features improve expressiveness or make  
code easier to write or change.

Participant 7

Lambda expressions come very handy in Qt's signal and slot connections, the  
code is more compact in such places - see the mentioning of c++11 lambda  
expressions on <https://wiki.qt.io/New_Signal_Slot_Syntax>.  
  
range based for-loops also allow for a more compact code. Same for the auto-  
keyword, especially when dealing with enums inside of class namespaces or so -  
this is where the auto-keyword can save a lot of space and a lot of typing  
work.

Participant 8

Speaking personally, I don't chase modern c++ features, \*per se\*,  but  
is a more a case by case choice where it can improve code quality and  
readability.  
A big factor is to reduce boilerplate code, to get the code much more  
readable (for instance range for instead of plain iterators.. at least  
where possible)  
speaking about iterators, also something like auto it =  
myLisy.begin(); is much more readable than QList<QString>::iterator ii  
= myList.begin();  
(on the other hand there are cases where i find auto to be a bit  
confusing when the type of the variable is not obvious at all, again,  
tere is no one size fits all)  
  
In KDE and Qt, there is a very strong use case for lambdas: Qt has  
this signal/slot mechanism, where for instance a button has a clicked  
signal where the code can react to that button being clicked.  
if the management code of the signal is very compact, one can connect  
to the signal completely inline with a lambda with something like  
  
connect(myButton, &QButton::clicked, this [this] () { // click  
management code }); without having to define a new method that would  
"pollute" the API.

Participant 9

My main motivation for adopting modern C++ features is simpler and more  
readable source code.  
  
There are several types of modern C++ features which make the source code more  
readable.  
  
C++11 auto: Compare  
  `QSortFilterProxyModel\* proxyModel = new QSortFilterProxyModel(this)`  
vs  
  `auto proxyModel = new QSortFilterProxyModel(this)`  
=> Less repetition, much easier to read.  
  
Lambda expressions enable you to embed for instance one-liner functions right  
to the point where it's used. For instance:  
  `for\_each(points.begin(), points.end(), [](Point const& p) {  
      cout << p->x();  
   });`  
=> Before you would have needed a separate function for this, with lambdas  
it's more compact.  
  
Same goes with range-based-for or default initializers for class members [1].  
It just results in much more readable/compact code in the end. You can find  
tons of examples in the internet which prove this.

Participant 10

The main motivation obviously is whether it makes sense. If the thing I'm using adds something.

In the case of lambdas and range-for is something that is basic of the basics in Java, I try to program in a more modern way, like in the case of KDE which needs a lot of callback, I have to pass a lot of callback, it's much better to declare the callback at the time that I'm passing it. In general you will only declare this function once. It makes more sense to declare the callback when you are passing it, without the need to declare it up there or in another header. Not to mention that lambda gives you a very elegant form. For example, if I want to store a value, I don't have to create an object, store that value and pass it to the function, it has a very simple way of holding state in lambda. So that's it. range-for is much better than using a counter in the for loop. It's something that should be basic when you iterate with arrays, with sequential things. These are things that make it more readable, closer to modern languages. And there is a third issue which is the Toolkits. Qt which is in KDE, but also GTK in the case of gnome. They make everything in a way that is simpler for you if you use these resources. So you adapt to the toolkits that evolve naturally with the language. So those are the motivations.

About the third option, which is Auto. This is already a personal choice. Regarding auto, I particularly don't like it. I think it makes it a little unreadable and you don't know clearly which type you are manipulating. This is to say that in the end it is a matter of choice. But it is a choice very much driven by the toolkits and the facilities that those lambda-like things offer.

Participant 11

Some of them make the code more readable. In particular, I am a fan of smart pointers, because they allow me to see memory ownership \*much\* easier. These features prevent some types of memory leaks. I have very little time to invest into KDE, and therefore I can only do very simple tasks, like cleanup/modernization.

**b) How often do you conduct software modernization efforts? Do you use any tools to support this kind of effort?**

Participant 1

Depending on the feature. C++11 was a huge improvement over C++99, which meant that a dedicated modernization effort was warranted. Tools that we used at this time were refactoring scripts[1] written by other KDE developers.

[1] https://invent.kde.org/sdk/kde-dev-scripts

More recent versions of the C++ standard offered smaller, more incremental changes. Thus, the modernization of code has also become more incremental for me. If I see code that would benefit from newer features while working on it, I take a note and do a follow-up commit to the change that I actually worked on. I'm not using any tool specifically for these smaller modernisations.

Participant 2

We use extensively clang-tidy and everything that LLVM project has to offer. :)

Participant 3

You should read the answer to question c first :)

It depends on the project mainly. Most of the time, a major Qt upgrade (Qt4=>Qt5, Qt5=>Qt6) will require a more modern C++, thus allowing the full feature set. But there are some projects that will simply use these features when they consider most distributions will have a modern compiler handling them. A lot of KDE projects will use tools like Clazy and Clang-tidy to help identify parts of code that need modernizing. Clazy is really Qt oriented while, as you surely know, Clang-tidy is a generic tool.

Participant 4

Rarely. And usually I don't use any tools. Often you need to look and think, so can't be easily automated with a tool. E.g. Q\_FOREACH can't be just replaced with range-for (https://www.kdab.com/goodbye-q\_foreach/), you have to be careful what is const and what is not const.

Though on the other hand for 0->nullptr conversions there are some tools (and compilers are good at pointing that too), similarly for override keywords.

Often code modernization just happens by writing new modern code. Occasionally when doing major porting, e.g, Qt4 to Qt5, or Qt5 to Qt6.

However, I know that some people do tend to focus more on modernization and do it more frequently.

Participant 5

Hard to say. I usually end up modernizing a particular piece of code when I'm fixing some bug or implementing a feature there and I notice that the particular area of code base could benefit from a little bit of modernization.

When KDE adopted C++11 we used some scripts to automatically port from the old connect() syntax to the new syntax [0], although regarding latest C++17 features (e.g. optional), those are hard to port automatically as usually change in code logic is needed.

Participant 6

Usually there is no organized modernization effort in KDE. People tend to start to modernize their own projects when they learn about the new possibilities, and if I remember correctly there were modern C++ talks at Akademy, which certainly helped adoption.

Many KDE developers are also subscribed to Planet KDE, which sometimes contains blog posts on modern C++ features, for example from KDAB.

Personally, I usually modernize code when I learn about the new C++ features.

On the tooling side, we are pretty well equipped, with clang-tidy and clazy. I use both regularly, and I think the same is true for many other KDE community members.

They are able to modernize some patterns automatically. clang-tidy is useful for general C++ modernization, while clazy handles Qt specific things.

Participant 7

We had some phases were we spent dedicated time on our side to modernize the code a bit. We used clang-tidy to identify and to adjust the code (https:// www.kdab.com/clang-tidy-part-1-modernize-source-code-using-c11c14/) , more was done later manually in multiple further iterations from time to time. Right now these activities are more or less done, we follow our code style using the c++11 constructs and there is no need anymore for a "modernization effort".

Participant 8

it's more continuous and low intensity, new code tends to use more modern c++ features, old code is ported more when it really needs to be rewritten. When the branching of the new major version of frameworks will happen (together with exclusive support of Qt6) this will probably change and become more thorough

Participant 9

Unfortunately applying software modernizations efforts usually requires convincing product management about its usefulness. We see that a lot in customer projects. Usually it's not high on their agenda b/c it does not bring the product "forward" in terms of stability / feature-richness for the next release. It's only something which will pay out in the long-term, maintaining that software. So in a business context it's relatively difficult to get time/ budget for doing these kind of things, usually it's being done when the code quality has degraded quite a bit already, i.e. when it's too late or people finally get too annoyed working with the legacy code base.

We use a lot of tooling in that area, most notably the Clang Tools. There's clang-tidy, a Clang-based linter / automatic refactoring tool, which is super useful for automated porting of some C++ source code bases: https://clang.llvm.org/extra/clang-tidy/

Clang-tidy has a huge list of "checks" which can be enabled, some checks actually have the ability to rewrite your source code files to "fix" the source code. There is a set of "modernize" checkers, for instance, which is basically rewriting source code to use more modern C++11/14/... C++ constructs. I.e. you can make it replace `Foo\* foo = new Foo(...)` with `auto foo = ...` automatically (with "modernize-use-auto"), or you can let it use default initializers for member variables automatically (with "modernize-use- default-member-init": this one's quite powerful but useful!). The full list of checks is here: https://clang.llvm.org/extra/clang-tidy/checks/list.html

There's also a Qt specific tool, called clazy, which is able to modernize Qt- based source code. Also based on Clang Tooling infrastructure: https://github.com/KDE/clazy

Both tools are integrated in IDEs, i.e. QtCreator, where users can modernize their source code by just clicking some buttons. Quite magic.

Participant 10

Unfortunately in Amarok I'm almost out of time, I had a son and it's been a year and a bit that I'm just releasing, I'm not touching the code itself anymore, Amarok is a little bit numb and I'm just releasing new versions. In the case of the linux client here at the company, I have been doing it weekly, because like, here I have been using GTK instead of Qt, but it is the same. Because GTK has evolved, the language has evolved and I want the client to be more readable, I do it weekly. In the end, if you don't do this, nobody will be interested anymore, the software will get old and will die, just like I am seeing with Amarok, so this work has to be done periodically or it will die. So, this work has to be done periodically or it will die. It is very difficult, the degree of entry and people look at it and say "gee, the work is so big that it is not worth trying. The tool that I use is an editor, an IDE which is Clion, I don't know if you have heard of it.

No I haven't, if you can send me the link here in the chat.

Have you heard of Intellij? it's from the same company. And from time to time I also use Kdevelop, which is from KDE, but so, as much as I like open source, kdevelop is one level below Clion. Not that Clion is better, but it is below the level of other tools from the company, but it is better than Kdevelop which I also use. From time to time I try to give Kdevelop a hand there, but for time reasons I have to be pragmatic.

extra question: Does Clion show you where you could modernize or does the tool actually go and make changes to your code?

No, it shows, I could, have the automated functions but I don't trust them. Even more in C + +, in Java, I would even trust it, but C + + is a slightly more complicated language I would say, pointer hierarchy issues, things that it is difficult for you to have a more correct static analysis. Since I did a Ph.D. in the area of parsing, I know that Java is much simpler in this respect. So he just hints, but overall he hints at very pertinent things. In general, it gives a list of two or three things, everything goes from removing things you don't use, like Const where you don't need them, to suggesting switching to lambdas.

Yes, there are several things like, includes that are unnecessary and are in your code and it warns that it may be included before and has several nice suggestions.

Participant 11

Irregularly. Whenever I feel like it and have a bit of time. No.

**c) Is there any direction from the KDE community about when features of a new version of the C++ language should be used more widely in KDE projects, or is this an individual or small team decision in each project?**

Participant 1

This is a team decision in each project. Of course, one looks at other projects (especially the frameworks libraries) as a reference, but ultimately every team can make their own rules.

Participant 2

We tend to coordinate such decisions across projects so Plasma devs can easily switch between projects without worrying about project specific conventions and policies, e.g. by discussing it in the plasma-devel mailing list first. One such example is preference of C++ smart pointers (std::unique\_ptr, std::shared\_ptr) over Qt smart pointers (QScopedPointer, QSharedPointer).

Participant 3

There is no major direction given. Each project does as it wants. For instance, calligra chose for a long time to keep compatibility with some very old libraries because one of our known users, Jolla, is still using Qt 5.6, prohibiting some new constructions from being used. But the KDE Frameworks will have a bigger influence, and they are much more carefully driven than the applications. If one of the major framework was to require, say, coroutines (in the future obviously), it sure would influence the applications into using this feature even for things not related to the said framework.

Participant 4

It's mostly individual decisions. However, there is also baseline determined by Qt requirements. E.g. Qt5 needs at least C++11 and Qt6 needs C++17, so all KDE software can depend at least on C++11 or C++17 if it targets Qt6.

Participant 5

For the "flagship" projects from the KDE community, like KDE Frameworks and Plasma, the community has agreed on the oldest supported OS/compiler, which also dictates the highest version of C++ that can be used. Individual smaller projects under the KDE umbrella usually have individual rules. Once a particular version of the C++ standard is allowed, there's no restriction on which features may be used, as long as it's supported by all compilers on all major target platforms.

We allowed ourselves to switch to C++11 once Qt with C++11 support was released, but since then switch to C++14 and 17 have been pushed by individual members when they really really wanted to use some of the features. Since the Qt API is mostly frozen in C++98/C++03 era (and C++11 support in Qt can mostly be reduced just to auto and lambdas), there's little incentive for the community to actively use latest C++ features.

Participant 6

For projects that have a stable interface and that projects outside of KDE depend on, the C++ standard is usually centrally decided. This is for example the case with the KDE frameworks.

For individual applications, this is the decision of the maintainers and teams

Participant 7

The KDE Community is working on multiple bigger project like the Frameworks, Plasma but also such big applications like Krita, Kdenlive, LabPlot, Cantor, etc. There are code styles and minimal compiler requirements within every single project and there is no "global" prescription for what to use. The project maintainers decide on their own what they need and when. I'm mainly working on LabPlot and Cantor and we have c++11 as the minimum supported version now.

Participant 8

Nowdays the modernization effor on c++ go a bit hand in hand with another effort as we are in the middle of the transition between Qt5 and Qt6, which will also bring a major, source and binary incompatible major release of the KDE frameworks, that leads to a general modernization effort.

Participant 9

extra-cmake-modules ships KDECompilerSettings.cmake which is usually included in every KDE project out there. That means these projects get the CMAKE\_CXX\_STANDARD set (here: C++17). The standard set in KDECompilerSettings.cmake is updated every now and then, after some discussion on some KDE mailing list (kde-devel, kde-core-devel) weighting the pros and cons.

Noone is being forced to \*use\* every language feature of a newer C++ standard however. This is a teams/projects decision. Usually volunteers have not a lot of time to do these kind of things, but they usually like modernizing their code as it makes their life easier in the long-term. Plus sometimes there are people (like me) who go "visit" multiple projects at a time and run the automated porting tools over them and modernize source code of multiple projects at one time. Since it's cheap to run the tools and verify the results afterwards; it just needs someone to actually do it and file a merge-request.

Participant 10

That I couldn't tell you for sure, because first of all KDE, although it is connected to Amarok. Amarok is in what people call an extra, which is a KDE application, but it's not a supported package, it's something that is in the middle of the road, so there I have all the freedom and not to mention that the project is kind of dying, it became too big, too hard to maintain and I joined just because I liked the player and I saw that the project was dying. For a year and a half I gave it a gas and tried to modernize it to try to attract a lot of people. In general it attracts a lot of people to thank you and say it's nice that you keep the project alive, but nobody exactly to put their hand in the code.

So there is no guidance like that. I know of some meetings that I have attended from KDE that they try to promote, but I couldn't tell you if there is a direction that is use C++ 11 or C++ 13 that I can't tell you. I have attended conferences like Akademy which is their main conference, that they try to promote new languages like Rust and things like that, so I suppose they encourage you to incorporate newer things with the caveat that versions have to be maintained. It has to be compatible, if you are developing for KDE 5 or Plasma 5 which uses Qt 5 or Qt 6, I used Qt5, you have to use whatever is compatible with Qt. Other than that the version of C++ and such, I had no direction, you have to be interested in getting newer things that break compatibility. So you better support things that are up to four years newer.

So as far as that, you have what they call LTS, which is Plasma LTS, I think for that there is some guidance, but I can't tell you what it is. You look for Plasma LTS and they tell you what the LTS version is, the version of QT that they're going to have some assurance there.

Participant 11

I am not really involved with the KDE community, therefore I can't really tell.

**d) We have found that some features for concurrent programming in modern C++ (like thread declaration, futures, and async) are little used within the KDE projects. Is there an explanation for this?**

Participant 1

Mostly, Qt already has facilities for concurrent programming. Moving from these to different ones has a far higher cost as with the features mentioned in the first question. I would therefore expect the uptake to be slightly higher in newly written applications. Is that supported by your data?

For KPhotoAlbum (the project that I maintain), I would only consider changing existing concurrent code to a different paradigm or newer concurrency facilities if the benefits of the change were substantial and/or if I needed to rewrite the code in question anyways.

Participant 2

I can't comment on this as my little playground is kwin and we use Qt APIs exclusively there, i.e. QThread and QFuture and things as such. Such preference is motivated by the fact that QThread is integrated with other Qt APIs better than say std::thread.

Participant 3

Again, the answer is Qt. With time, some C++ features are promoted as replacement for Qt features. But some features have been integrated in Qt for a very long time, and the newcomers have a hard time rising above and replacing years of habits. For instance, std::thread. It's a welcome addition to the C++ standard library, and it was badly needed. But the need for an abstraction to system threading APIs is not new, and for a long time Qt had a QThread class, deeply rooted in its object and event models. If you are writing a Qt application, std::thread is more cumbersome to use than QThread.

Participant 4

I don't know. Maybe because KDE uses Qt API for threads and asynchronous stuff, so a lot of signal/slots, etc... And also it's more advanced stuff, so naturally used less.

Participant 5

I think that's because those features have been provided by the Qt framework long before they were standardized in C++ (e.g. QThread for std::thread, QFuture for std::future, QtConcurrent is sort of an equivalent to std::async) and they generally integrate better with the rest of the Qt-based codebase than their modern C++ counterparts.

However, majority of asynchronous programming in Qt/KDE is done through the Qt's event loop and I guess most of the use-cases in KDE don't require a dedicated thread.

Participant 6

This can probably be explained by Qt having it's own concurrency primitives that are sometimes incompatible with what the standard library offers.

Multi-threading is mostly done by the means of QThread and QObjects. Methods of QObjects can be executed indirectly by the Qt metaobject system, and that is clever enough to call functions from the QThread which the object belongs to.

At least Qt's mutexes are compatible with the standard library's mutex lockers.

For futures and async, there is a Qt framework (QtConcurrent) which provides similar APIs that are easier to integrate into existing Qt projects.

Participant 7

I think the reason for this is the very good support for concurrent programming in Qt:

https://doc.qt.io/qt-6/threads.html

https://doc.qt.io/qt-6/qtconcurrent-index.html

This is available in Qt already for very long time and many projects, including LabPlot and Cantor, are using this. Right now there is no need and no plans on our side to move away from what Qt is offering. This can change in future, of course, but right now our spare time is better invested in other areas.

Participant 8

Qt in the past had library level features for all of those things as were missing from the base language, so we use a lot of those features from Qt (QThreads, QFutures or a slightly different approach in KDE frameworks as KJob) we will eventually slowly move to standard c++ versions of those, but will take time, as porting things that are working is not the immediate priority at the moment. for instance it's not long we started to use more of the stl standard containers as std:vector, typle and so on rather than QVector, QList and so on.

Participant 9

Using threads in uncontrolled manner makes a program indefinitely harder to debug and extent, so if you don't necessarily need them you should try to stay away from them. Qt makes heavy use of the signal-slot mechanism which allows you to do "async" programming without using any concurrent programming constructs yourself. Best examples are the classes from the QtNetwork module: There you can realize networking related functionality without ever using threads or futures, since you can use Qt's signal-slots mechanism to get notified about incoming connections, about data available to read, connection state changes, etc. pp..

See for instance: https://doc.qt.io/qt-6/qtnetwork-fortuneserver-example.html

I guess that's one major reason. There are however some of the more complex KDE projects which make heavy use of concurrent programming, most notably the KDevelop IDE and Krita (which I know top off my head) which are quite heavy on the computational side of things. I.e. KDevelop needs to scan source code in the background all the time, Krita needs to run algorithms on images, both run that on multiple cores in the background.

There surely are lots of others KDE projects which also make use of it.

Participant 10

Yes, it does, first of all these things came much later in C++ than they did in Qt. You won't find these names.on top of Qt there is a thing called KDElibs, they implemented all of this or reused stuff from Qt. As a toolkit it already provides this, you won't find it in C++ or else because it is desktop applications, by the nature of desktop applications you need very little of it. In desktop most of the stuff is an event loop, which will show you and that's it. Everything that will already come in Qt, we never depended on C++.

It was for a long time, and since Qt is already a level up, it provides that in a very good way for you to manage it. If you want to fire twenty threads at most, you have that control already built in, so it doesn't make much sense, the toolkit already provides that and it doesn't make sense to change what is already good.

I will compare with Java, this came from the language, I don't use any toolkit on the side, I am talking about the backend. This has been in the language since very early on. C++ took too long, it came in C++ 2017, something that in 2007, 2008 Java already had. Then the toolkits arrived and not only Qt, there is Boost, STL, and basically what C++ did was to incorporate what these two provided. And when you already have the toolkit you don't need anything else.

A classic example is threads, Qt already provided Qthread and in addition KDE developed Kthread on top of Qthread which provided some more. Everything is already there, you don't need anything else.

Participant 11

I don’t know