The Practical Programmer's Guide to C++20

Giuseppe D'Angelo and Ivan Čukić

If you're a C++ programmer, it probably won't have escaped your attention that the C++20 standard has been released and is supported (in great part) by the most popular compilers like GCC, Clang, MSVC, and Apple Clang. We're excited by this not only because we love C++ and because many of us at KDAB are language standards wonks, but because it provides some of the most revolutionary changes to the language since the ground-breaking C++11 release. C++20 provides C++ with even more power and expressiveness, and levels the playing field between this veteran workhorse language and newer upstarts.

However, a list of the new C++20 features often sounds like a rules-lawyer's minutiae. While the standard contains dozens of improvements and fixes, we're going to focus on the changes we think will make the biggest differences to the everyday C++ programmer. Of course, we'll be talking in detail about the biggest four improvements (concepts, coroutines, modules, and ranges), but we'll also share a couple of smaller changes that we think are interesting.

A concept specifies the template author's expectations for how the template should be used.

Concepts

Simply put, a concept specifies the template author's expectations for how the template should be used. In C++20, the author adds concepts to define the valid uses of that template – that is, what types, functions, or other characteristics are required to use the template properly.

Let's see what this looks like in action by using the well-worn factorial, the "hello world" of functions that every programmer is probably tired of seeing written out in code. We apologize for bringing one more version of factorial into the world, but bear with us, because it will turn out to be handy for explaining more about concepts in a bit.

```
#include <concepts>
#include <iostream>

std::integral auto factorial(std::integral auto a){
    if (a <= 0) return 1;
    else return a * factorial(a - 1);
}

int main() {
    std::cout << factorial(10) << std::endl;
    return 0;
}
```

Example 1: Factorial with concept

As you can see, this is not too different than a plain-old factorial function, although we use **std::integral** instead of an integer type. **std::integral** is a concept, and it tells the compiler that as long as the parameter conforms to a standard integer type (like any of the built-in 8-, 16-, 32-, or 64-bit signed or unsigned types), go ahead and accept this invocation of the function template as valid. In turn, since we use a concept-enforced **std::integral auto** as the return type, it ensures our return type will also be an integer class.

Concepts turn our simple function into a generic one that handles all integral types, which makes it more flexible and reusable. Concepts will also let us easily extend our function to other integer or even non-integer types as we'll soon see.

The compiler knows exactly what is expected and can produce much more meaningful error diagnostics when constraints aren't met.

Concepts and improved errors

One big benefit of using concepts manifests itself when we start using the template above. The compiler knows exactly what is expected and can produce much more meaningful error diagnostics when constraints aren't met. For example, here's what happens when you try to use the function template above with something that's not an integer.

```
main.cpp: In instantiation of 'auto [requires ::Integral<<placeholder>, >]
factorial(auto:11) [with auto:11 = double]':
main.cpp:50:33:   required from here
main.cpp:27:30: error: use of function 'auto [requires ::Integral<<placeholder>, >]
factorial(auto:11) [with auto:11 = double]' with unsatisfied constraints
   27 |     else return a * factorial(a - 1);
      |                     ~~~~~~~~~^~~~~~~
main.cpp:25:15: note: declared here
   25 | Integral auto factorial(Integral auto a){
      |               ^~~~~~~~~
main.cpp:25:15: note: constraints not satisfied
main.cpp: In function 'int main()':
main.cpp:50:33: error: use of function 'auto [requires ::Integral<<placeholder>, >]
factorial(auto:11) [with auto:11 = double]' with unsatisfied constraints
   50 |     std::cout << factorial(-10.5) << std::endl;
      |                            ^
main.cpp:25:15: note: declared here
   25 | Integral auto factorial(Integral auto a){
      |               ^~~~~~~~~
main.cpp:25:15: note: constraints not satisfied
main.cpp: In instantiation of 'auto [requires ::Integral<<placeholder>, >]
factorial(auto:11) [with auto:11 = double]':
main.cpp:50:33:   required from here
main.cpp:8:9:   required for the satisfaction of 'Integral<auto:11>' [with auto:11 =
double]
main.cpp:9:26: note: the expression 'std::is_integral<_Tp>::value [with _Tp = double]'
evaluated to 'false'
    9 |     std::is_integral<T>::value;
      |                          ^~~~~
```

Example 2: Concept-simplified error reporting

Egads, that's a lot of compiler output to wade through. We've highlighted the key parts that let us know what's going on, but it's not too difficult to spot the problem. We failed constraints based on the Integral concept and that's because the class we tried to use didn't conform to **std::integral**.

Without concepts, the compiler can't generate a human understandable error by the time a problem is encountered deep down in a template's guts.

Without concepts, the compiler can't generate a human understandable error by the time a problem is encountered deep down in a template's guts. In an equivalent version of this example without concepts, the compiler spits out 172 lines of complaints for a similar one-line error. Those messages say nearly nothing about the true nature of the issue or where it was encountered by the programmer – they're mostly about template argument deduction or substitution failures and mismatched non-derivative iterators that are template-mangled beyond recognition. You can eventually work your way back to the root cause of all these errors, but it's far less obvious what the issue is, and it takes substantially more time to spot the issue.

By cleaning out much of the compiler error clutter and focusing on the root problem, concepts can make using templates for everyday programming less intimidating.

Concepts can make using templates for everyday programming less intimidating.

Concept overloads

What if we want to extend our ubiquitous factorial to handle real and complex numbers? That's easy too. Concepts are great at creating different overloaded behaviors. We can have one version for integral types while we have other versions for non-integral types that will avoid writing a lot of duplicated boilerplate that would otherwise be required. This is the next big benefit of concepts, since overloading of generics can be specified in a clean and declarative way, rather than relying on non-obvious compiler behaviors like SFINAE.

```
#include <iostream>
#include <complex>
#include <concepts>
using namespace std::complex_literals;

// while waiting for P2078 to add the is_complex type
trait,
// we implement a makeshift version...
template<typename T>
concept Complex = requires(T a, T b) {
        { a.imag() };
        { a.real() };
        { a + b } -> std::convertible_to<T>;
};

template<typename T>
concept Continuous = Complex<T> || std::floating_point<T>;

std::integral auto factorial(std::integral auto a){
    if (a <= 0) return 1;
    else return a * factorial(a - 1);
}
```

Overloading of generics can be specified in a clean and declarative way, rather than relying on non-obvious compiler behaviors

```
std::floating_point auto factorial(std::floating_point auto
a){
    decltype(a) one = 1.0;
    return std::tgamma(a + one);
}

Complex auto complexGamma(Complex auto a){
    auto z = 0.0 + 0i;
    // A whole mess of complex math here...
    // (for one example, see https://www.johndcook.com/
blog/cpp_gamma/)
    return z;
}

Complex auto factorial(Complex auto a){
    decltype(a) one = 1.0;
    return complexGamma(a + one);
}

int main() {
    using namespace std::complex_literals;
    std::cout << factorial(10) << std::endl;
    std::cout << factorial(-10.5) << std::endl;
    std::cout << factorial(10.0 + 2i) << std::endl;
    return 0;
}
```

Example 3: Generic factorial with concepts

If more than one constraint matches a function, template, overload, or argument, the compiler chooses one that is more specialized..

We've introduced the **requires** keyword to enforce the constraints our complex concept needs, specifically what function signatures your generic code depends on. In this case, we require that any class calling itself complex must support real(), imag(), and addition. Clearly, moving this from an illustrative example to functional code might require a lot more constraints in our **requires** clause. We've also created a Continuous concept in the code above that, although not used in this example, shows how concepts can be simply composed of other concepts with Boolean operators.

Concept specialization

Multiple concepts for the same type are handled by the compiler in a process called partial ordering of constraints. Fundamentally, this means that if more than one constraint matches a function, template, overload, or argument, the compiler preferentially chooses one that is more specialized before one that is less specialized. This is a formalized way to specify to the compiler the behavior that a programmer might intuitively expect.

The "tighter" the concept is defined, the higher it will be prioritized.

A quick illustration shows how this works.

```
using namespace std::complex_literals;

// Definition #1 – handles floating types only
void f(std::floating_point auto x);

// Definition #2 – handles floats and complex numbers
void f(Continuous auto x);

int main() {
    f(3.14f);                              // Case A:
compiler calls f #1
    f(0.707 – 0.707i);                     // Case B:
compiler calls f #2
    return 0;
}
```

Example 4: Concept specialization

The "tighter" the concept is defined, the higher it will be prioritized when the compiler has more than one competing option that matches. So, any type matching the std::floating_point concept (like 3.14f in case A, as well as any other float or double) will match both definitions of function `f`. However, because definition `#1` is more specialized – technically speaking, `f #1` is subsumed by `f #2` – the compiler will use definition `#1` for floating point numbers. In case B, the only concept that applies to complex numbers is our custom Continuous concept, so the compiler chooses function definition `#2`.

Concepts bring the promise and power out of the hands of library authors and into the hands of everyday programmers.

Concepts make templates easier

Concepts allow you to create compile-time pseudo-duck typing that expresses the template author's intent up front during function invocation, rather than relying on compiler trickery or waiting until problems are encountered during template instantiation. We feel this clear and readable communication helps brings some of the promise and power of generic template techniques out of the hands of library authors and into the hands of everyday programmers.

Ranges

Next on the C++20 hit parade are ranges, which can be thought of as "Unix pipes brought to C++". Because they don't require loops and because range views are generally value-based (non-mutating), there's less room for error – no off-by-one errors or memory allocation issues. Ranges bring a bit of the bliss of functional programming to C++. Let's see them in action.

Range views don't have off-by-one errors or memory allocation issues.

```
#include <ranges>
#include <iostream>

int main()
{
    auto ints = std::ranges::iota_view{1, 33};
                                    // Half-closed range,
32 is last

    auto even_bytes = [](int i) { return (i % 8)==0; };
                                    // Keep only the byte
boundaries

    auto largest_value = [](int i) { return (1ULL<<i)-1;
};
                                    // Biggest expressible
unsigned

    for (uint64_t i : ints |
                      std::views::filter(even_bytes) |
                      std::views::transform(largest_
value)) {
        std::cout << i << ' ';
    }

    std::cout << std::endl;
}
```

Example 5: Range example source

Ranges don't have to be lazy, but the great part is they can be.

The output of this program is a list of the biggest unsigned values that can fit in a value that's from one to four bytes long.

```
255 65535 16777215 4294967295
```

Example 6: Range example output

Admittedly, there are countless better ways to do this contrived example. However, it allows us to quickly show how easy it is to construct a function by composing several lightweight range operations.

As one example, the iota_view function doesn't generate an array; its iterator generates as many numbers as you need and only when they're needed. (Ranges don't have to be lazy, but the great part is they can be.) This laziness allows ranges to address situations where the entire problem set can't be loaded into memory at once or when you're dealing with generators of infinite (or nearly infinite) length.

Because ranges are lightweight – much of their magic is handled

Ranges don't impose the big memory allocation burdens or run-time costs you might expect from a functional programming paradigm.

by the compiler at compile-time – they aren't imposing big memory allocation burdens or run-time costs that you might anticipate from a functional programming paradigm.

The power of ranges

Great, but can we use ranges to solve real-life problems? Sure! A classic example is counting document word frequencies, inspired by the infamous battle of titans Donald Knuth and Doug McIlroy. In the original 1986 Programming Pearls column published by the ACM, Knuth's 10 pages of procedural Pascal was replaced by McIlroy using a six-line Unix script. The big advantage in simplicity comes from conceptualizing the problem as a series of functions pipelined together.

We build our text word counter through a composition of several simpler, pre-existing functions.

We can do much the same with C++ ranges, by building our text word counter through a composition of several simpler, pre-existing functions. Our C++ range variant is nice and tight. The pipe syntax of ranges is also very familiar to anyone who's ever used Unix or Linux, making it clear to read even for the uninitiated.

```
#include <iostream>
#include <vector>

#include <range/v3/view.hpp>
#include <range/v3/action.hpp>
#include <range/v3/view/istream.hpp>
#include <range/v3/range/conversion.hpp>

using namespace ranges;

auto string_to_lower(const std::string &s) {
    return s | views::transform(tolower) |
to<std::string>;
}

auto string_only_alnum(const std::string &s) {
    return s | views::filter(isalnum) | to<std::string>;
}

bool string_is_empty(const std::string &s) {
    return s.empty();
}

int main(int argc, char *argv[])
{
    const int n = argc <= 1
                  ? 10
                  : atoi(argv[1]);

    const auto words =
              istream_range<std::string>(std::cin)
              | views::transform(string_to_lower)
```

```
                | views::transform(string_only_alnum)
                | views::remove_if(string_is_empty)
                | to_vector | actions::sort;

    const auto frequency = words
                | views::group_by(std::equal_to{})
                | views::transform([] (const auto &group) {
                        const auto size        =
distance(group);
                        const std::string word =
*cbegin(group);
                        return std::pair{size, word};
                    })
                | to_vector | actions::sort;

    for (auto [count, word]: frequency | views::reverse
                                       | views::take(n)
            ) {
        std::cout << count << " " << word << '\n';
    }

    return 0;
}
```

Example 7: Using ranges to build an index

Unfortunately, there's a catch, and that is, the code above won't run with out-of-the-box C++20 alone.

Wow – powerful stuff. Another good example that might pique your interest is a CppCon presentation where Eric Niebler shows how he uses ranges to build a calendar app with simple, understandable, bug-free, and highly reusable code.

Realizing ranges

Unfortunately, there's a catch, and that is, the code above won't run with out-of-the-box C++20 alone. The code here is using Eric's implementation of ranges that unfortunately, to prevent delaying the standard, was only partially adopted by the standards committee. While ranges in their fullest expression are extremely powerful, the C++20 version is lacking some key capabilities like group_by, zip, concatenate, enumerate, and lots more that you'd need for many pretty common uses.

The good news is that you can experiment around with the ranges that C++20 does provide today. If you get hooked and want a bit more power, you can download working and tested range-v3 code while you're waiting for more range expressiveness to make its way into the standard. Hopefully by C++23, a much more comprehensive set of range functionality should be able to clear the standards committee.

Modules make it simple to reuse code – add an import statement and you're done.

Modules

We next look at modules. There's not a lot to say here, other than it's about time. C++ has its reusability model based around header files, while other programming languages have slightly cleaner ways of reusing code. Modules add the simplicity of reuse that Python, Go, Java, or C# users enjoy – simply add an import statement and you're done.

```
// Note: the standard library isn't yet module-ready in
C++20,
// so this example won't compile. If it was, it might look
like this:

import <iostream>;

int main()
{
        std::cout << "Hello module world!\n";
}
```

Example 8: Importing a module

An increasing proportion of code is located within headers, driven by template-based libraries, external components, and internal application reuse.

What's the big deal – isn't this the same as #include? Kind of, but better.

An increasing proportion of code is located within headers, driven by the popularity of template-based libraries like Boost, Eigen, and Asio, by the total number of external components being used, and by the demand for internal application reuse. Forcing the compiler to lexically insert an entire universe of header files into the source only serves to bring compile times to a crawl. Precompiled headers help some, but only when used carefully. They don't really solve the problem; they only attempt to work around it.

Is it easy to create a module using C++20 syntax? Absolutely trivial.

```
// simplelib.cpp

export module simple.example;
           // dots can be optionally inserted for module
name readability

export void foo(){}     // foo() is accessible by outside
world
void bar(){}            // bar() is only visible within
this file
```

Example 9: Exporting a module

Modules are embraced by modern C++ paradigms: no file guard hacks needed, fewer dependency issues, and faster compilation times.

What modules bring

Modules are a cleaner and more capable solution to the header file that are embraced by modern C++ paradigms: no file guard hacks needed, fewer dependency issues, and faster compilation times. Modules also promise effective shielding of private functions, more consistent initialization order guarantees, and better global optimization. There's quite a bit more to C++20 modules than we've explored here – interface, implementation, header units, global and private module fragments – here's a nice and concise summary.

Modules promise effective shielding of private functions, more consistent initialization order guarantees, and better global optimization.

What modules miss

Unfortunately, there are two downsides to modules. The first is the chicken-and-egg problem: until C++20 gets more widespread adoption and people start using modules, there will be little incentive for library authors to provide them. And without common libraries using modules ... you see the problem. This is compounded by the other big issue (at least of this writing), which is compiler support. None of the top compilers fully support modules per the C++20 specification just yet. (And as the comment in our example points out, at the current time, the standard doesn't even fully embrace modules in the standard library.)

We'll also likely need a diversity of C++ programmers in the community stressing modules in real-world applications before all of the kinks get worked out. Headers have well-known capabilities (and quirks) as well as decades of backwards compatibility that will keep them a part of everyday C++ usage for the foreseeable future. But it's nice to look forward to the cleanliness of modules, even if it's only within your own internal projects for the near-term.

Coroutines

The last big thing that C++20 brings to the party is coroutines. Have you ever done any of the following?

- Constructed your own event-driven framework
- Created lazy generating classes
- Interleaved several timing-dependent functions within a single-threaded design
- Written tokenizers and parsers that read their data asynchronously

In all these cases – and many more – coroutines come to the rescue.

Coroutines solve the nightmarish mess of function pointers, callback routines, and state management that can result from solving these sorts of problems. Any situation that you might normally have to solve with callbacks are a natural fit for coroutines. So is anything with asynchronous I/O or functions that maintain lots of internal state management.

Any situation that you might normally solve with callbacks are a natural fit for coroutines.

What does this look like in code?

```
generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}
```

Example 10: Using coroutines to implement the iota generator

A lazy iota generator is a great example of the power of coroutines since it can be implemented almost trivially. iota is a function that generates an endless number of sequential integers, and the implementation is just about as simple as you could imagine. Sit in a while-forever loop, and continually spit out bigger numbers. The compiler takes care of return and reentry, state management, and function pointer manipulation. Sure, you could do this in regular old C++ too, but it'd be far messier, uglier, and error prone.

What about an asynchronous I/O example?

```
task<> tcp_echo_server() {
  char data[1024];
  for (;;) {
    size_t n = co_await socket.async_read_
some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

Example 11: Echo server with coroutines

Again, clean and simple.

C++20 coroutines shouldn't be thought of as a multi-threading replacement.

Coroutines aren't threads

Although C++20 coroutines allow the interruption of long-running functions, they shouldn't be thought of as a multi-threading replacement. They aren't at all. Rather, they are somewhat like non-preemptive multitasking – they require full cooperation on behalf of the entire program they're in to operate properly. But as this example shows, there can be a bit of overlap between the two concepts.

If our echo server example were any more complex, you might be able to justify using multi-threaded code. In this case though, the cooperatively threaded model provided by coroutines does the trick. It's "nearly multi-threaded" without having to worry about spinning up threads, adding locks or synchronization primitives, or figuring out how to cleanly tear down the thread. And that's the case for an awful lot of event-driven things in our programs. Multithreading takes far more work to "do it right" to justify using it in every case. Coroutines can make it easy to introduce a level of simple "parallel" execution without worrying about callbacks or state machines. Just remember it's not intended to replace real multi-threading and don't get carried away.

Coroutines can make it easy to introduce a level of simple "parallel" execution without worrying about callbacks or state machines.

Coroutine with care

Even though you don't need special locks or access controls, coroutines do introduce a few restrictions. You're not allowed to have varadic arguments, plain return statements, or placeholder return types (like auto or Concepts). You also can't have constexpr, constructors, destructors, or the program's main() function be coroutines. All-in-all though, this is a pretty short list of exclusions that we can definitely live with.

If there's any drawbacks to the new coroutines, it's only that they're not better supported in the C++ libraries. The required necessities are there in the language in the form of **co_await**, **co_yield**, and **co_return**, and they're very usable in that form if you want to roll-your-own code. However, it's a bit disappointing that there aren't as many building blocks or library types available to help roll them out properly. Similarly to ranges, we expect this situation to improve as some community proposed features get rolled into standard libraries for C++23. If you want a preview of the power of coroutines being expressed to its fullest in the meantime, you might want to check out the CppCoro library.

std::format gives us safe string formatting that complements the existing C++ methods of string output.

Anything else?

We've talked about the biggest four changes in C++20, but there are two other smaller yet still impactful changes that are worth pointing out.

std::format

The C++ community has two options for creating formatted strings: the archaic past of printf/sprintf strings that are both cryptic and ultra-dangerous, or modern stream operators that are terrible for localization. Thankfully, C++20 has adopted Victor Zverovich's fmt library in the form of std::format, which solves both of these issues. This is because **std::format** gives us safe string formatting that complements the existing C++ methods of string output.

Be jealous of Python string formatting no more.

By using placeholder-based formatting and keeping the value formatting options within the (more easily localized) format string, std::format is i18n-approved and simple to use. Be jealous of Python string formatting no more.

```
std::cout << std::format("{1}, {0}!\n", "world", "hello");
std::cout << std::format("he{0}{0}o, {1}!\n", "l",
"world");
```

Example 12: Two ways to generate "hello world!" with positional args

Three-way comparison operator

One last nicety is the three-way comparison or <=> operator, otherwise known as the "spaceship" operator for its rough resemblance to a space invaders sprite. This operator does for any type what good old strcmp has done for decades: communicates less than (negative value), equals (zero), and greater than (positive value) status in a single call.

What's the big deal? The great part about spaceship is that it makes writing your own operators much easier. It's a bit of a pain to supply all six comparison operators for your own classes (`<`, `<=`, `==`, `>`, `>=`, `!=`). That becomes especially true if you provide comparisons from your class to other built-in or library classes (and vice versa), when the number of comparison operators can explode. Thankfully, now you just need to write one spaceship operator for your class, and one spaceship for each comparison type. That's it – two trivial operators instead of 18 for the below example.

```cpp
#include <compare>
#include <iostream>

class ExValue {
 public:
    constexpr explicit ExValue(int val): value{val} { }
    auto operator<=>(const ExValue& rhs) const = default;
    constexpr auto operator<=>(const int& rhs) const {
        return value - rhs;
    }
private:
    int value;
};

template<typename A, typename B>
const auto whatis(A a, B b) {
    auto comparison = (a <=> b);
    if (comparison < 0)
        return "less";
    else if (comparison > 0)
        return "greater";
    else
        return "equals";
}

int main() {
    std::cout << whatis(ExValue(10), ExValue(12)) << std::endl;
    std::cout << whatis(ExValue(12), ExValue(10)) << std::endl;
    std::cout << whatis(ExValue(8), ExValue(8)) << std::endl;
    std::cout << whatis(10, ExValue(12)) << std::endl;
    std::cout << whatis(100, ExValue(50)) << std::endl;
    std::cout << whatis(ExValue(10), 40) << std::endl;
    std::cout << whatis(ExValue(40), 10) << std::endl;

    std::cout << (bool) (ExValue(10) < ExValue(15)) << std::endl;
    std::cout << (bool) (ExValue(10) > ExValue(15)) << std::endl;
    std::cout << (bool) (ExValue(10) <= ExValue(15)) << std::endl;
    std::cout << (bool) (ExValue(10) >= ExValue(15)) << std::endl;
    std::cout << (bool) (ExValue(10) == ExValue(15)) << std::endl;
    std::cout << (bool) (ExValue(10) != ExValue(15)) << std::endl;
}
```

Example 13: Simplification provided by spaceship

We've even made it more complicated than it absolutely needs to be when comparing our class against int by using a subtraction. We could have easily used <=> here instead, which would read more clearly. However, we've used subtraction as a bit of a reminder that underneath the implementation of spaceship, the logic can often simply boil down to a trivial and natural operation.

default allows the compiler to synthesize a comparison in many simple cases.

Also note that we didn't even have to write our type's spaceship function – using **default** allows the compiler to synthesize one in many simple cases. We get all 18 operators for "free" because the compiler now converts all comparison operations into spaceship automatically. When the compiler sees **A < B**, it silently interprets that as **(A <=> B) < 0**, and similarly transforms any other comparison operator.

Spaceship provides more flexibility behind comparisons too. It has different ordering categories like strongly ordered, weakly ordered, or partially ordered. These allow you to sort, filter, or partition based on comparisons where objects may be equal but not equivalent or to introduce special values like NaN or Infinity that can fail any comparison. (In fact, these special orderings are why we don't just print out the results of **<=>** directly in the code above – it doesn't just return a simple int.) But it's the simplification of user-created types that is the biggest change added by the three-way comparison operator that most programmers will directly notice.

Three-way comparison results in two trivial operators instead of 18 in our example.

Summary

The new standard has many changes that will mostly be of interest to library builders and language experts, things like **consteval**, **constinit**, **no_unique_address**, lambda function improvements, and others. These all help to improve the foundation of the language, make code more efficient or easier to write, close gaps in the standard, or lay groundwork for upcoming changes in C++23. But the six additions that we talk about in this whitepaper are the ones we think will make the biggest difference in the lives of practical programmers.

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com