How C++ lambda expressions can improve your Qt code

In case you've missed it, lambda expression support has been added to C++ in C++11. In this article we are going to see how to take advantage of lambda expressions in your Qt code to simplify it and make it more robust, but also which pitfalls to avoid.

But first, what is a lambda expression?

Lambda expressions are anonymous functions defined directly inside the body of another function. They can be used everywhere one currently passes a pointer to a function.
The syntax of a lambda expression is the following:

```
[captured variables](arguments) {
    lambda code
}
```

Ignoring the "captured variables" part for now, here is a simple lambda which increments a number:

```
[](int value) {
    return value + 1;
}
```

We can use this lambda in a function like `std::transform()` to increment the elements of a vector:

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
```

```
    std::vector<int> vect = { 1, 2, 3 };
    std::transform(vect.begin(), vect.end(), vect.begin(),
                   [](int value) { return value + 1; });
    for(int value : vect) {
     std::cout << value << std::endl;
    }
    return 0;
}
```

This prints:

```
2
3
4
```

Capturing variables

A lambda expression can make use of variables in the current scope, by "capturing" them.
Here is an example where we use a lambda to compute the sum of a vector.

```
std::vector<int> vect = { 1, 2, 3 };
int sum = 0;
std::for_each(vect.begin(), vect.end(), [&sum](int value) {
    sum += value;
});
```

As you can see, we captured the local variable `sum` so that we could use it in the lambda. `sum` is prefixed with `&` to indicate that we capture *by reference*: inside the lambda, `sum` is a reference, so any change we make to it affects the variable outside the lambda.
If you want a copy of the variable instead, you capture it without prefixing it with `&`.
If you want to capture multiple variables, just separate them with commas, like function arguments.
It is not possible to directly capture a member variable, but you can capture `this`, and get access to all members of the current object.
Under the hood, the captured variables of a lambda are stored in an hidden object, unless the compiler can be sure the lambda is not going to be used outside the current local scope, in which case it can optimize this away by directly referring to the local variables.
It is possible to be a bit lazy and capture all local variables. To capture them all by reference, use `[&]`. To capture them all by copy, use `[=]`. We do not recommend doing so however, because it makes it too easy to reference variables whose life-cycle would be shorter than the life-cycle of your lambda, leading to odd crashes, even capturing by copy can cause such crashes if you copy a pointer. Explicitly listing the variables you depend on makes it easier to avoid this kind of traps. If you want to learn more about this trap, have a look at item 31 of "Effective Modern C++".

Lambdas in Qt connections

If you use new-style connections (and you should, because type-safety is good!), you can use lambdas on the receiving side. This is great for small dispatcher functions.
Here is an example of a phone dialer, where the user can enter a number and start a call:

```
Dialer::Dialer() {
    mPhoneNumberLineEdit = new QLineEdit();
    QPushButton* button = new QPushButton("Call");
    /* ... */
    connect(button, &QPushButton::clicked,
            this, &Dialer::startCall);
}
void Dialer::startCall() {
    mPhoneService->call(mPhoneNumberLineEdit->text());
}
```

We can get rid of the `startCall()` method using a lambda:

```
Dialer::Dialer() {
    mPhoneNumberLineEdit = new QLineEdit();
    QPushButton* button = new QPushButton("Call");
    /* ... */
    connect(button, &QPushButton::clicked, [this]() {
        mPhoneService->call(mPhoneNumberLineEdit->text());
    });
}
```

Using lambdas instead of QObject::sender()

Lambdas are also a great alternative to the use of `QObject::sender()`. Let's imagine our dialer is now an array of number buttons.
Without lambdas the code to compose a number could look like that:

```
Dialer::Dialer() {
    for (int digit = 0; digit <= 9; ++digit) {
        QString text = QString::number(digit);
        QPushButton* button = new QPushButton(text);
        button->setProperty("digit", digit);
        connect(button, &QPushButton::clicked,
                this, &Dialer::onClicked);
    }
    /* ... */
}
void Dialer::onClicked() {
    QPushButton* button = static_cast<QPushButton*>(sender());
    int digit = button->property("digit").toInt();
```

```
    mPhoneService->dial(digit);
}
```

We could use a QSignalMapper to get rid of the `Dialer::onClicked()` dispatcher method, but using a lambda is more flexible and leads to even simpler code. We just need to capture the digit associated with the button and call `mPhoneService->dial()` directly from the lambda.

```
Dialer::Dialer() {
    for (int digit = 0; digit <= 9; ++digit) {
        QString text = QString::number(digit);
        QPushButton* button = new QPushButton(text);
        connect(button, &QPushButton::clicked,
            [this, digit]() {
                mPhoneService->dial(digit);
            }
        );
    }
    /* ... */
}
```

Don’t forget object life cycles!

Look at this code:

```
void Worker::setMonitor(Monitor* monitor) {
    connect(this, &Worker::progress,
            monitor, &Monitor::setProgress);
}
```

In this slightly contrived example, we have a Worker instance which reports progress to a Monitor instance. So far so good.

Now suppose that the `Worker::progress()` signal has an int argument and we want to call a different method on `monitor` depending on the value of this argument. We can try something like this:

```
void Worker::setMonitor(Monitor* monitor) {
    // Don't do this!
    connect(this, &Worker::progress, [monitor](int value) {
        if (value < 100) {
            monitor->setProgress(value);
      } else {
          monitor->markFinished();
      }
    });
}
```

This looks good but… this code can crash!
Qt connection system is smart enough to delete connections if either the sender or the receiver is deleted, so in our first version of `setMonitor()`, if `monitor` is deleted the connection is deleted as well. But now we use a lambda for the receiver: Qt has no way to now that this lambda makes use of `monitor`. Even if `monitor` is deleted, the lambda will still be called and the application will crash when it tries to dereference `monitor`.
To avoid that, you can pass a "context" argument to the `connect()` call, like this:

```
void Worker::setMonitor(Monitor* monitor) {
    // Do this instead!
    connect(this, &Worker::progress, monitor,
            [monitor](int value) {
      if (value < 100) {
          monitor->setProgress(value);
      } else {
          monitor->markFinished();
      }
    });
}
```

In this version, we pass `monitor` as a context to `connect()`. It won't affect the execution of our lambda, but when `monitor` is deleted, Qt will notice and will disconnect `Worker::progress()` from our lambda.
This context is also used to determine whether the connection should be queued or not. Just like a classic signal-slot connection, if the context object thread is not the same as the code emitting the signal, Qt will use a queued connection.

Alternative to QMetaObject::invokeMethod

You might be familiar with a way to call a slot asynchronously using `QMetaObject::invokeMethod`. Given a class like this:

```
class Foo : public QObject {
public slots:
    void doSomething(int x);
};
```

You can tell Qt to call `Foo::doSomething()` when it is back to the event loop using `QMetaObject::invokeMethod`:

```
QMetaObject::invokeMethod(this, "doSomething",
                          Qt::QueuedConnection, Q_ARG(int, 1));
```

This works, but:

- The syntax is ugly
- It is not type-safe
- It forces you to declare methods to call as slots

You can replace this code with a `QTimer::singleShot()` calling a lambda:

```
QTimer::singleShot(0, [this]() {
    doSomething(1);
});
```

It is a little less efficient because `QTimer::singleShot()` creates an object under the hood, but unless you call this multiple times per second, the performance cost is negligible, so the benefits outweigh the drawbacks.

Similarly, you can specify a context before the lambda, this is mostly useful to jump between threads. Be careful though: if you are stuck with a version of Qt older than 5.6.0 there is a bug in `QTimer::singleShot()` which causes a crash when used between threads. We found out about that one the hard way...

Key takeaways

- Take advantage of lambda expressions to remove dispatcher methods when connecting Qt objects together
- When using a lambda in a `connect()` call, always define a context
- Do not capture more variables than necessary

That's it for this article, we hope you enjoyed it, now go and find places where you can replace boilerplate methods with fancy lambdas!