Introduction to Q#

Q# (Q-sharp) is a domain-specific and open-sourced programming language, part of Microsoft's Quantum Development Kit (QDK), used for expressing quantum algorithms. It is to be used for writing subroutines that execute on an adjunct quantum processing unit (QPU), under the control of a classical host program and computer.

Q# can be installed on Windows 10, OSX and Linux. The instructions to install Q# can be found in the online documentation here.

If you prefer not to install Q# on your local computer, you can use one of the machines in CSE's Virtual Lab found here. The Windows 10 machines already have .Net Core SDK, Visual Studio and VS Code installed to get your started, you should still install the Q# extension to get syntax-highlighting, code complete, etc.

To get help with Q# and the QDK, feel free to ask questions on our messages board, come to office hours as posted on the calendar, or ask in stackoverflow. The Q# team is constantly monitoring any questions posted there with the "q#" tag.

Writing Q# programs.

Operations and functions are the basic unit of execution in Q#. They are roughly equivalent to a function in C or C++ or Python, or a static method in C# or Java.

A Q# operation is a quantum subroutine. That is, it is a callable routine that contains quantum operations.

A Q# function is a classical subroutine used within a quantum algorithm. It may contain classical code but no quantum operations. Specifically, functions may not allocate or borrow qubits, nor may they call operations. It is possible, however, to pass them operations or qubits for processing. Functions are thus entirely deterministic in the sense that calling them with the same arguments will always produce the same result.

Together, operations and functions are called callables.

When you create a new Q# project (see online documentation); a new `Program.qs` file is created with this content:

```
namespace hw5 {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;


    @EntryPoint()
    operation SayHello() : Unit {
        Message("Hello quantum world!");
    }
}
```

To build and run this code, from the command line in the your project's folder do:

```
dotnet run
```

For example:

```
~/cs490q$ dotnet run
Hello quantum world!
```

As you can see, Q# is structurally very similar to familiar languages such as C# and Java in its use of semicolons to end statements, curly brackets to group statements, function calls and doubleslash to introduce comments. It is a strongly typed language so all variables, arguments and return values must have an associated type. Q# supports a familar set of classical types like:

- Int
- BigInt
- Bool
- Double

but also some quantum-specific:

- Qubit
- Result
- Pauli

The complete set of primitive types can be found in Q# type model documentation.

In Q#, |0⟩ and |1⟩ are represented by `Result.Zero` and `Result.One` accordingly and qubits can only be allocated inside an operation with the `using` statement. When it gets allocated a qubit is always in the |0⟩ state. To measure a qubit and read its value you use the `M` intrinsic operation.

As such, the next `HelloQuantum` operation always return `Result.Zero`:

```
    // Always return |0>
    operation HelloQuantum() : Result {
        using(q = Qubit()) {
            return M(q);
        }
    }
```

To invoke this operation at runtime, modify the @EntryPoint:

```
namespace hw5 {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;

    // Always return |0>
    operation HelloQuantum() : Result {
        using(q = Qubit()) {
            return M(q);
        }
    }

    @EntryPoint()
    operation SayHello() : Unit {
        let result = HelloQuantum();
        Message($"Result: {result}");
    }
}
```

Variables in Q# are defined using `let`. variables are immutable and can be used in inter-polated strings (i.e. strings that start with a dollar sign) to be printed out into the console.

And from the command line:

```
$ dotnet run
Hello quantum world!
Result: Zero
```

A more interesting example for a Q# operation is a quantum random bits generator. The following RandomBits operation returns a different value each time it is invoked:

```
    operation RandomBits(n: Int) : Result[] {
        mutable r = new Result[n];

        using (qs = Qubit[n]) {
            ApplyToEach(H, qs);

            for(i in IndexRange(qs)) {
                // other languages: set r[i] <- M(qs[i]);
                set r w/= i <- M(qs[i]);
            }

            ResetAll(qs);
        }

        return r;
    }
```

`RandomBits` demonstrates other Q# features:

- `mutable` is used to initialize variables that can be modified later in the code using `set`.
- `new Result[n]` is used to initialize a new `Result` array of size `n`.
- `[]` is used to access the element of an array.
- New arrays can be created from existing ones via copy-and-update expressions. A copy-and-update expression is an expression of the form `arr w/ idx <- value` that constructs a new array with all elements set to the corresponding element in `arr`, except for the element(s) at `idx`, which are set to the one(s) in `value`. The resulting array can be assigned to the same variable by using the `w/=` operator.

- Q# has a rich set of built-in libraries, for example:
  - `ApplyToEach`: is an Operation that receives another operation as paramter and an array, and applies the given operation to each element of the array. In this particular case
  - `IndexRange`: is a function that given an array, creates a range to iterate over the indices of the elements in the array.
  - `ResetAll`: similar to `Reset`, it makes sure all elements of a qubit array are in `Result.Zero` state so they can be safely de-allocated.

- All callables belong to a namespace. In Jupyter, you can use all the operations in the Microsoft.Quantum.Intrinsic and the Microsoft.Quantum.Canon namespaces automatically. To use operations in other namespaces, like `IndexRange` from `Microsoft.Quantum.Array`, you have to use the fully qualified

name of the operation (i.e. `Microsoft.Quantum.Array.IndexRange`) or include an `open` statement at the top.

Let's call RandomBits from a new @EntryPoint (the old @EntryPoint must to be deleted as a Q# program can only have one):

```
    @EntryPoint()
    operation CreateRandom(n: Int) : Unit {
        for(i in 1..10) {
            Message($"{i}: {RandomBits(n)}");
        }
    }
```

This new entry point receives a parameter: `n`; to provide a values to entry points include them in the `dotnet run` command after `--`, for example:

```
$ dotnet run -- -n 4
1: [One,One,Zero,Zero]
2: [One,One,One,One]
3: [Zero,Zero,One,One]
4: [Zero,Zero,One,Zero]
5: [Zero,One,One,One]
6: [Zero,Zero,One,Zero]
7: [Zero,One,One,Zero]
8: [Zero,One,Zero,One]
9: [Zero,One,Zero,One]
10: [Zero,One,One,Zero]
```

Q# can automatically calculate the adjoint and the controlled version of an operation. In the declaration you simply include `is Adj + Ctrl`. You can then invoke the operation's adjoint by using the `Adjoint` keyword. Similarly, you can invoke the quantum-controlled version of the operation using the `Controlled` keyword and passing an array of control qubits:

```
namespace hw5 {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Diagnostics;

   /// # Summary
    /// Given a qubit in |0⟩, prepares the qubit's state to |+⟩
    operation PreparePlus(q: Qubit) : Unit
    is Adj + Ctl {
        H(q);
    }

    /// # Summary
    /// Given a qubit in |0⟩, prepares the qubit's state to |-⟩
    operation PrepareMinus(q: Qubit) : Unit
    is Adj + Ctl {
        X(q);
        H(q);
    }

    @EntryPoint()
    operation AlwaysZero() : Unit
```

```
    {
        using((ctrls, q) = (Qubit[1], Qubit())) {

            Message("before:");
            DumpMachine();

            // The operation we just defined:
            PreparePlus(q);

            Message("after");
            DumpMachine();

            // Passing control qubits
            Controlled PrepareMinus(ctrls, q);

            // Undo everything we just did.
            Adjoint Controlled PrepareMinus(ctrls, q);
            Adjoint PreparePlus(q);
        }
    }
}
```

Invoking `AlwaysZero` yields the following output.

```
$ dotnet run
before:
# wave function for qubits with ids (least to most significant): 0;1
|0⟩:     1.000000 +  0.000000 i  ==     ******************** [ 1.000000 ]     --- [  0.00000 rad ]
|1⟩:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
|2⟩:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
|3⟩:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
after
# wave function for qubits with ids (least to most significant): 0;1
|0⟩:     0.707107 +  0.000000 i  ==     **********          [ 0.500000 ]     --- [  0.00000 rad ]
|1⟩:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
|2⟩:     0.707107 +  0.000000 i  ==     **********          [ 0.500000 ]     --- [  0.00000 rad ]
|3⟩:     0.000000 +  0.000000 i  ==                          [ 0.000000 ]
```

Notice the output generated by DumpMachine(). It prints the state (wave function) of the entire system it's a great tool to debug your quantum programs.

To learn more...

The online documentation is a great resource to learn more about Q#. and includes further information about:

- Q# file structure
- Operations and functions
- Variables
- Work with qubits
- Control flow
- Test and debug

The Q# team is constantly monitoring any questions posted there with the "q#" tag at stackoverflow.com. You can try there too.