Welcome!

#qsharp • @MsftQuantum

While everyone settles in, feel free to get started.

INSTALLING THE QUANTUM DEVELOPMENT KIT

1. Install .NET Core SDK
2. Install Visual Studio Code + Extension
3. Install Project Templates

RUNNING THE QUANTUM KATAS

1. Download the Quantum Katas
2. Open a kata folder in VS Code
3. Run `dotnet test`

INSTALL GUIDE
aka.ms/qdk-install

KATA DOWNLOAD
aka.ms/quantum-katas


Nitrogen fixation


Carbon capture


Materials science


Machine learning

Microsoft

Quantum Programming

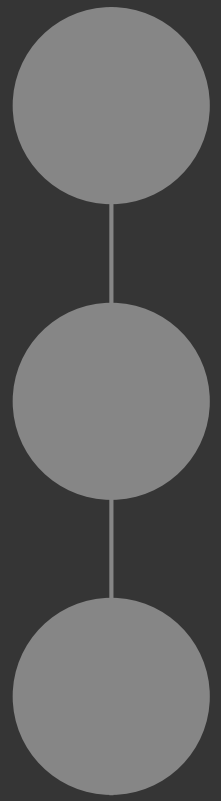
THE QUANTUM DEVELOPMENT KIT

Christopher Granade
Research SDE
chgranad@microsoft.com

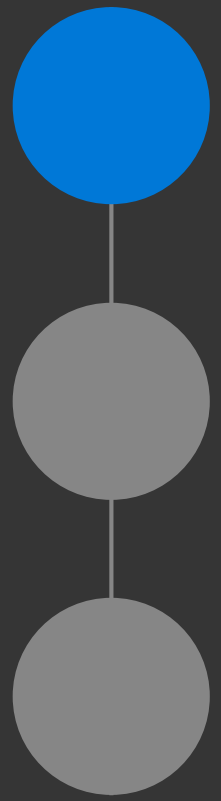
Bettina Heim
Senior SDE
beheim@microsoft.com

Quantum Architectures and Computation Group, Microsoft

Why Quantum Programming?



Why Quantum Programming?



Practical Cost Estimates

Example: Quantum Chemistry of Fe_2S_2

Used in reactions and energy transport in photosynthesis

3,000!

CLASSICAL ALGORITHM

INTRACTABLE

1

QUANTUM ALGORITHM (2012)

YEARS

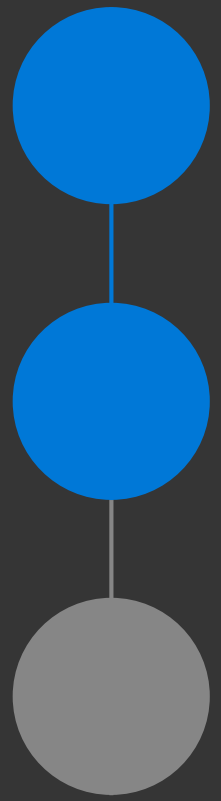
QUANTUM ALGORITHM (2015)

DAY

Progress made through

- Better quantum algorithms
- More complete cost estimates
- Improved classical statistics

Why Quantum Programming?



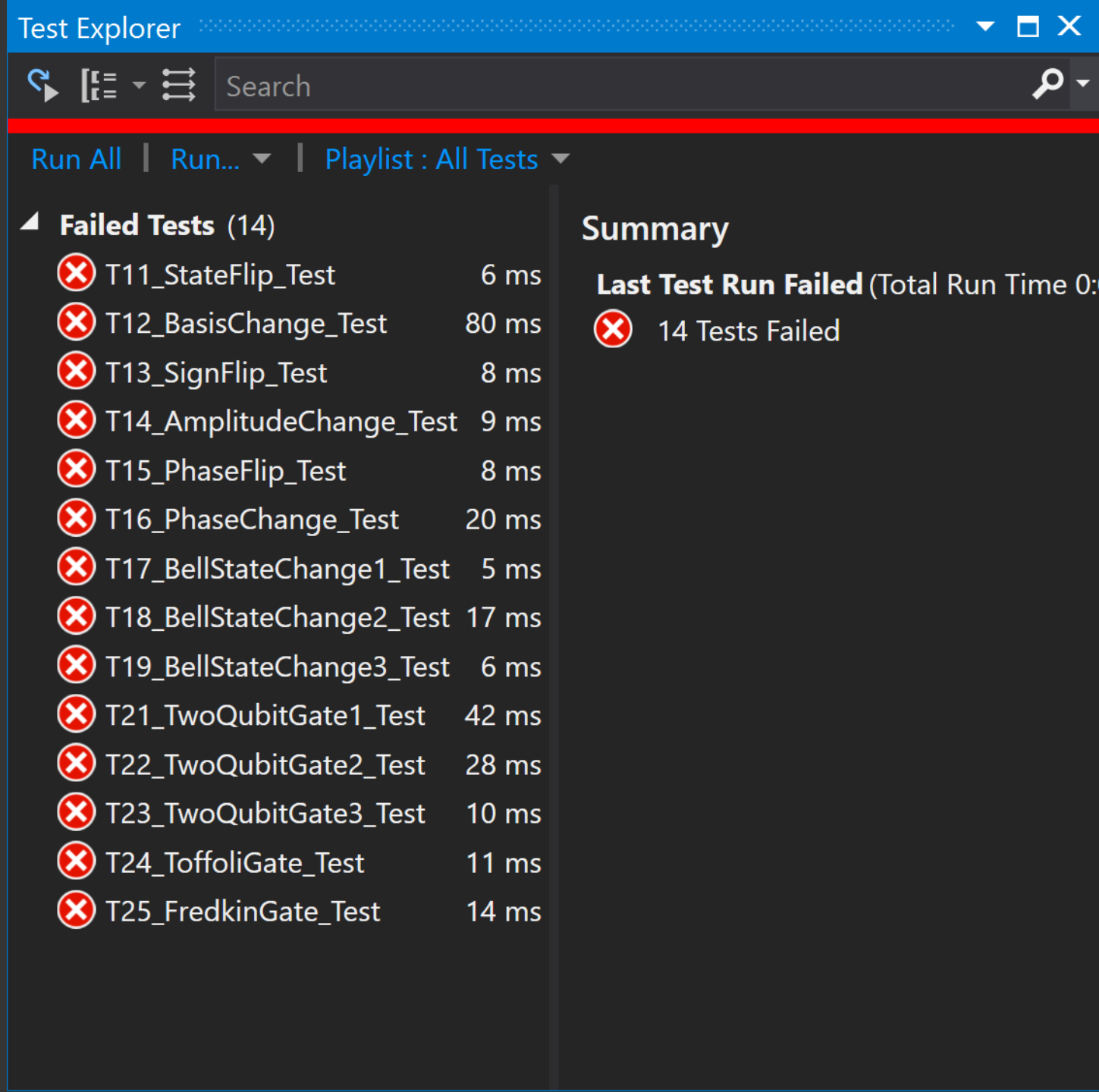
Practical Cost Estimates

Debugging Quantum Algorithms

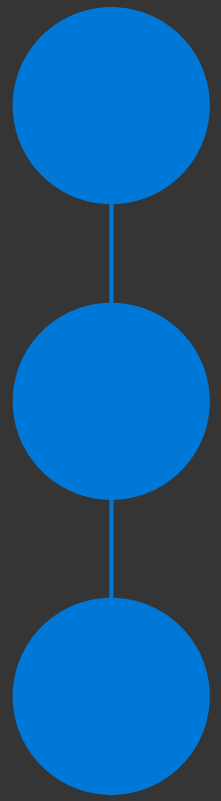
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

—Brian Kernighan

Even harder to debug an algorithm if you can't run it.



Why Quantum Programming?



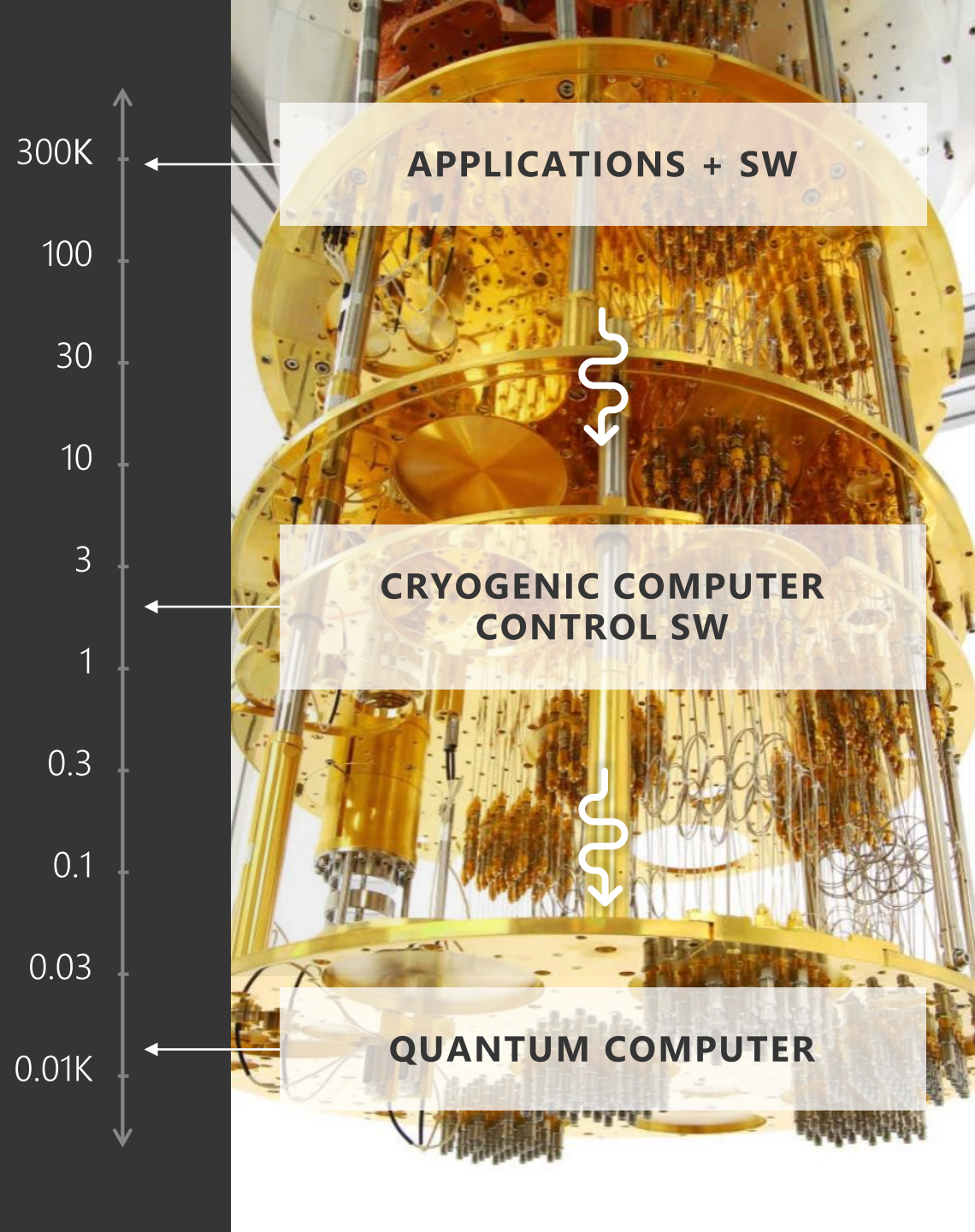
Practical Cost Estimates

Debugging Quantum Algorithms

Algorithms to Quantum Applications

Computing at every level

Developing applications takes a **full-stack** approach, including both classical and quantum logic



The Quantum Development Kit

Q#

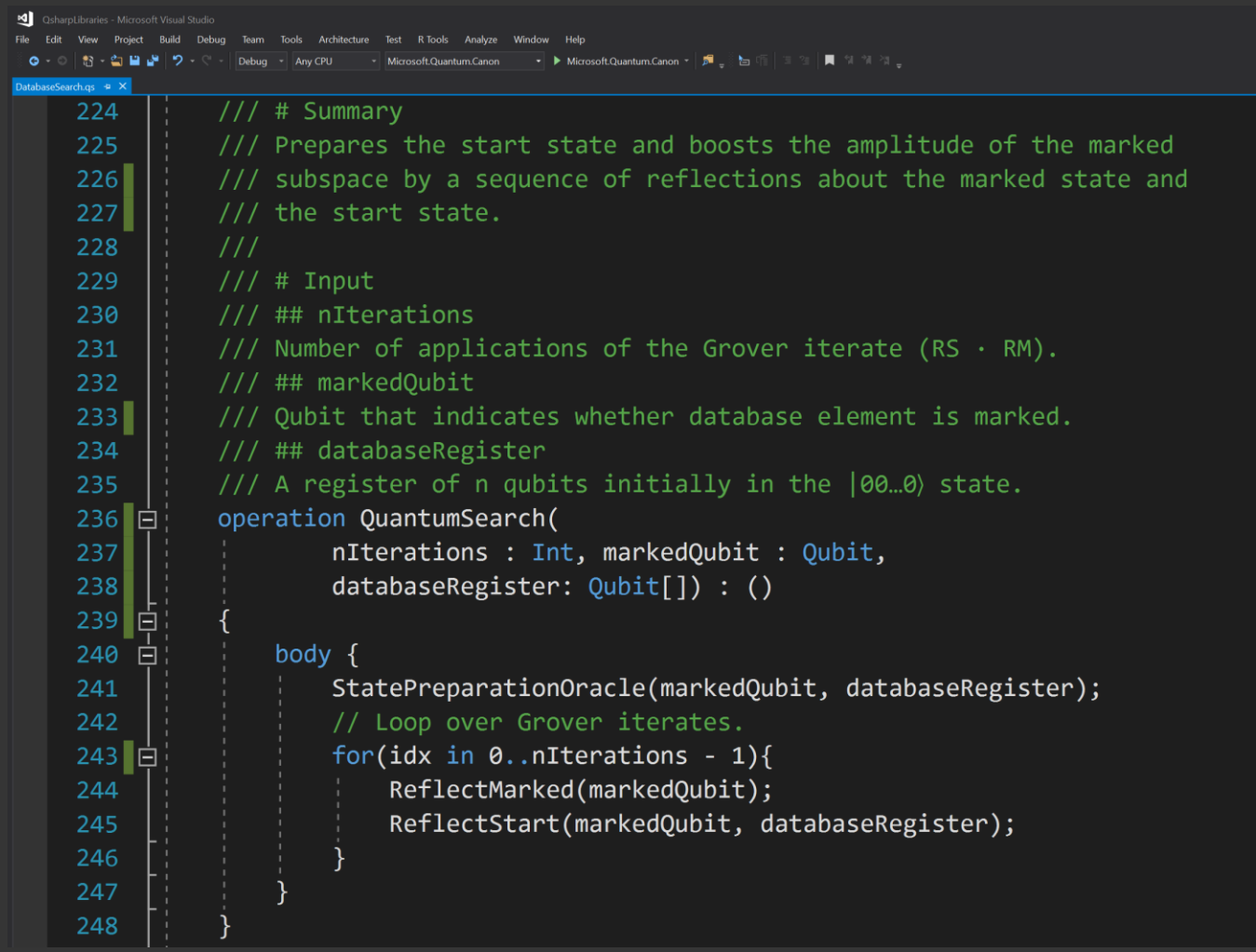
A new language for quantum *algorithms*.

Open-source libraries/samples

Phase estimation, amplitude amplification, state preparation, and more.

Powerful dev tools

Extensions for Visual Studio and Visual Studio Code.



The Quantum Development Kit

.NET Core

Works with C#, F#, and VB.NET.

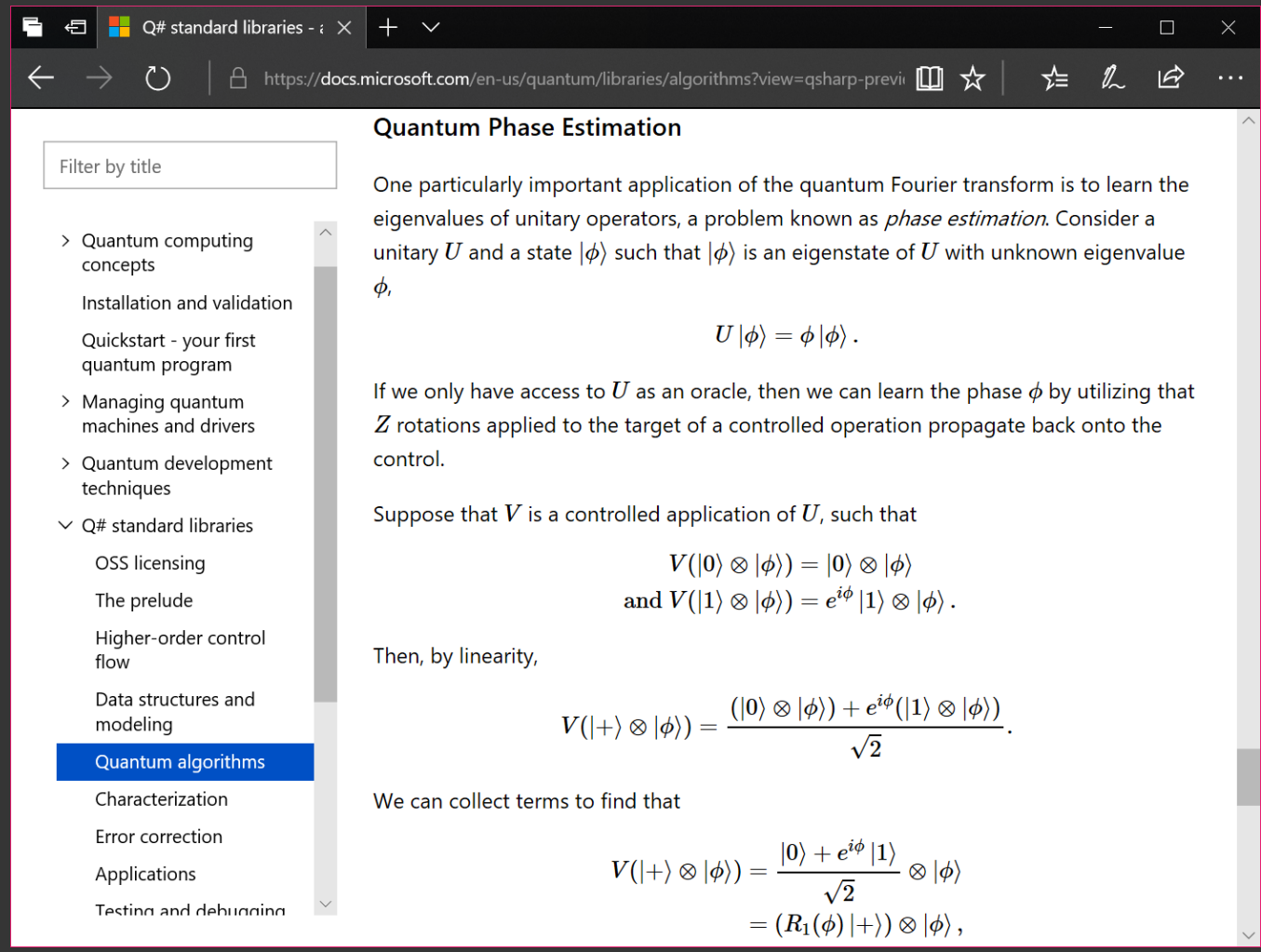
Cross-platform

Windows, macOS, and Linux.

Versatile simulation

Local simulator and cost estimator.

Comprehensive documentation



Why Q# Works

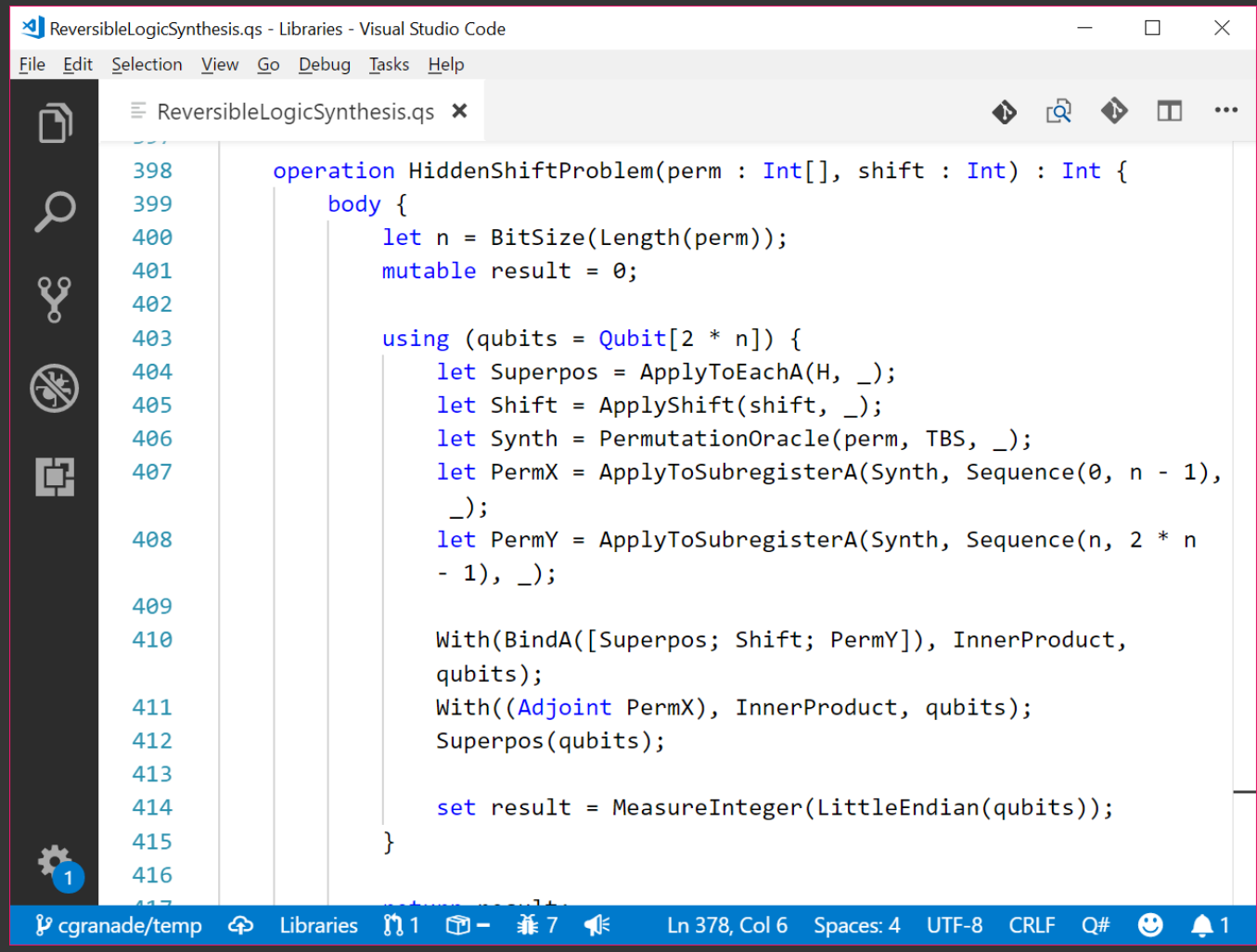
High-level programming

First-class quantum operations, partial applications, type parameters, and more.

Rich Standard Libraries

Classical logic

Easy to implement hybrid quantum/classical algorithms and adaptive measurement protocols.



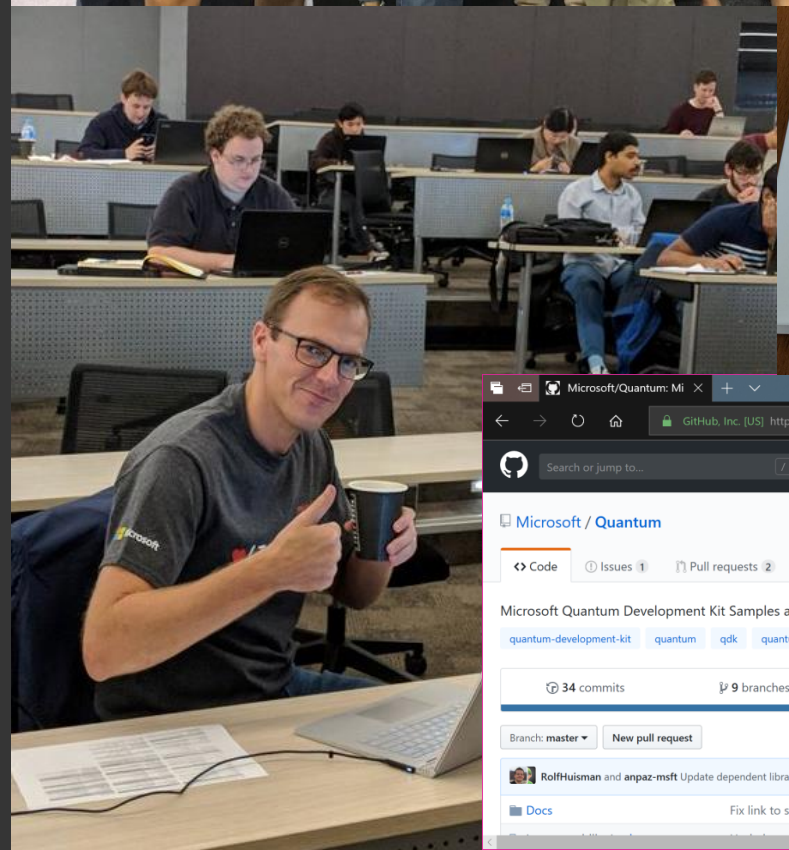
The Quantum Development Kit Community

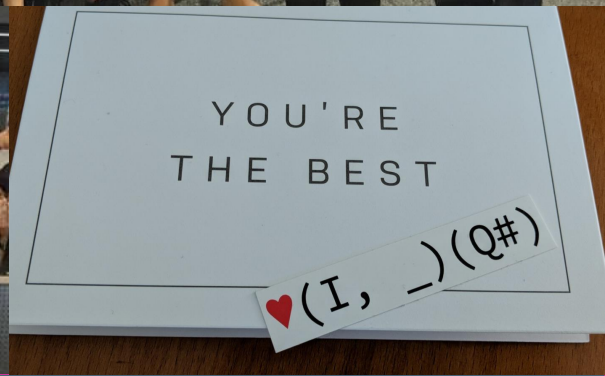
Feedback
quantum.uservoice.com

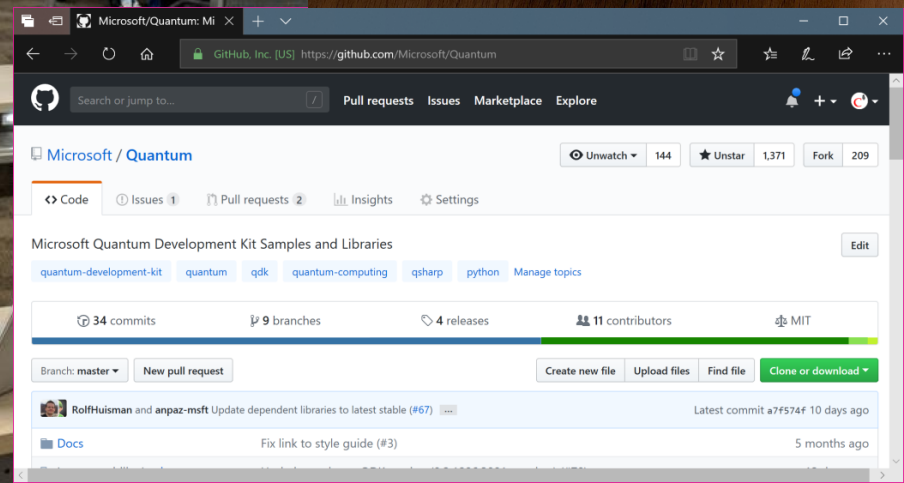
Documentation
github.com/MicrosoftDocs/quantum-docs-pr

Open Source Contributions
github.com/Microsoft/Quantum

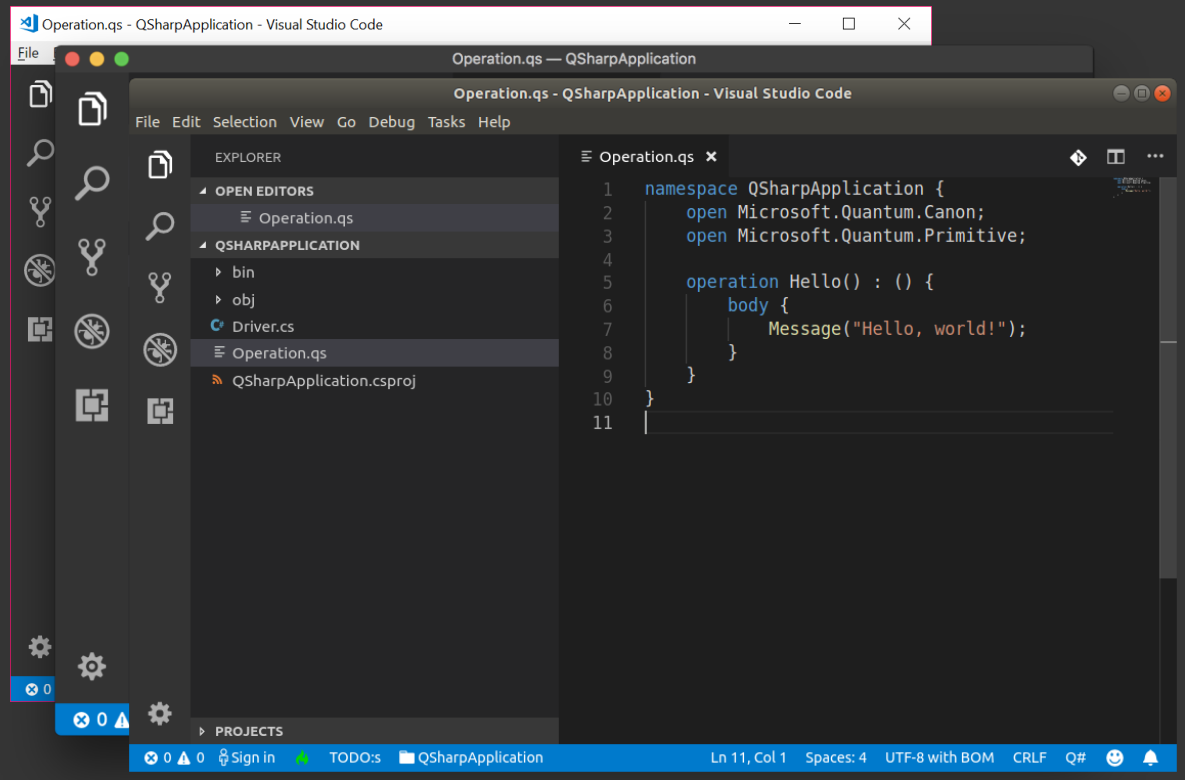
Social Media
#qsharp • @MsftQuantum







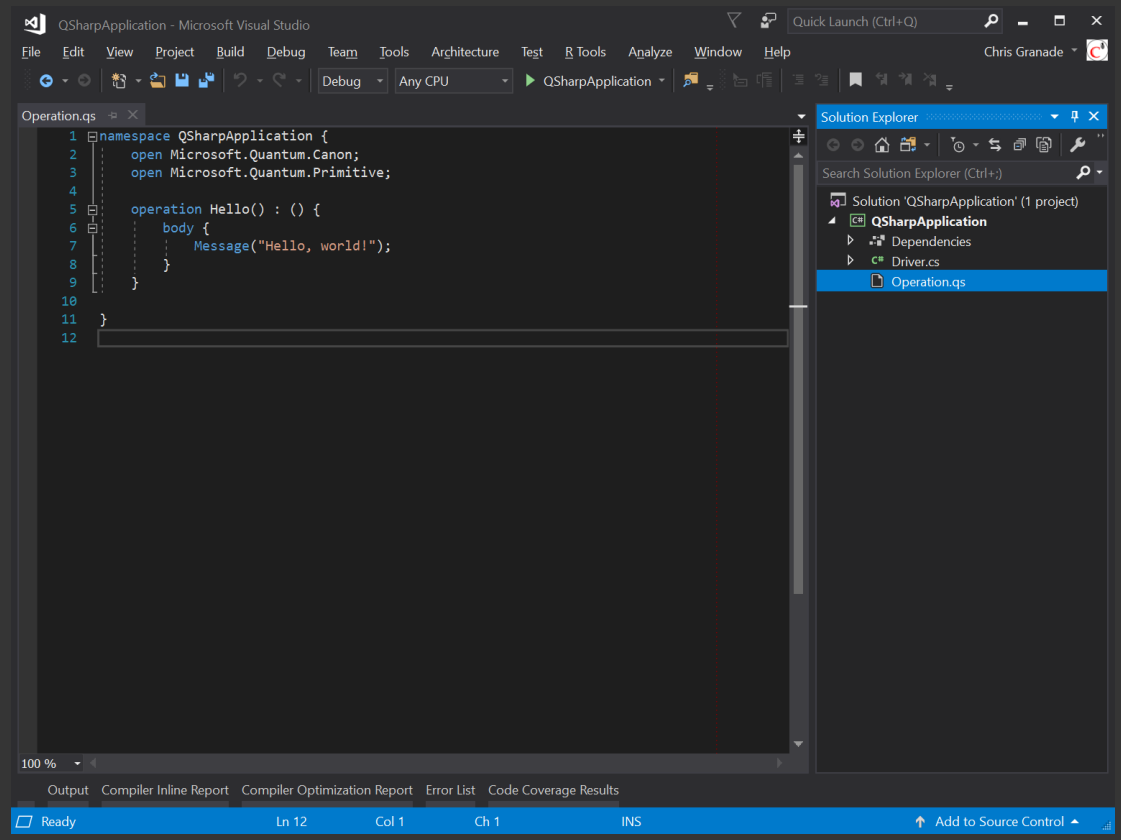
Visual Studio Code



Open source for Windows 10, macOS, and Linux.

Q#, C#, F#, C++, Python, Julia, LaTeX, Bash, PowerShell, TypeScript and more.

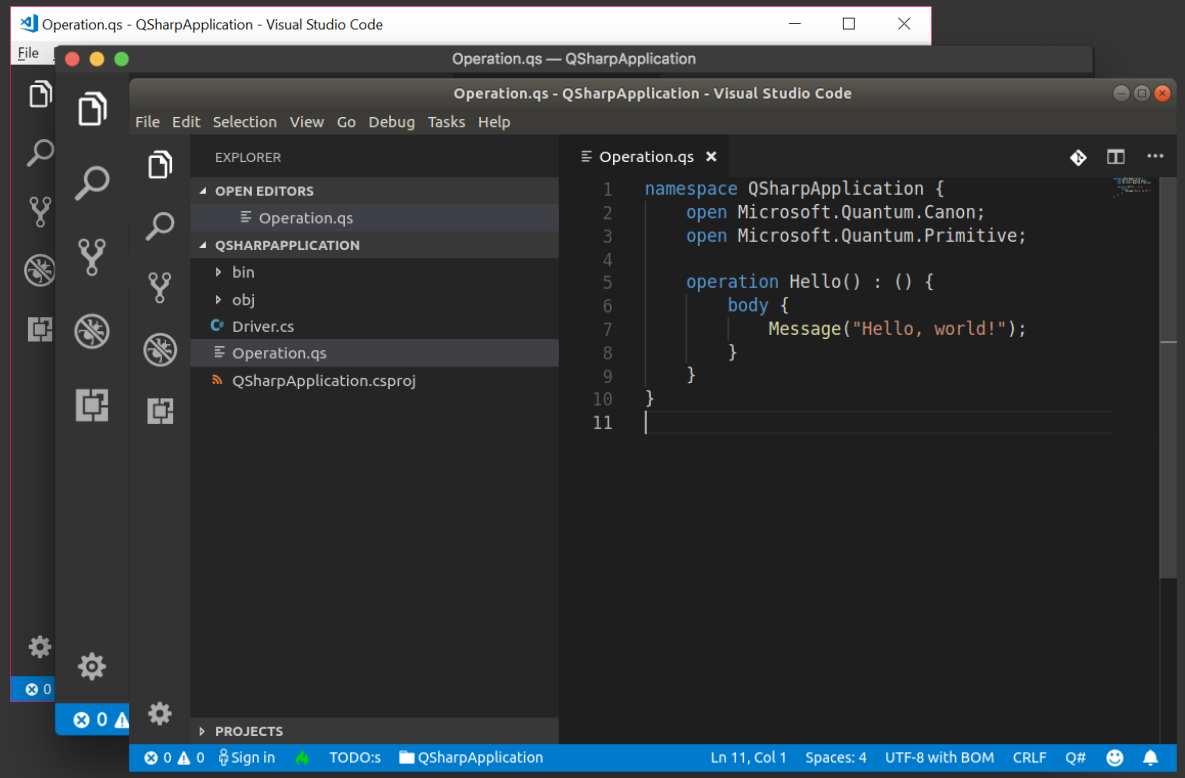
Visual Studio 2017



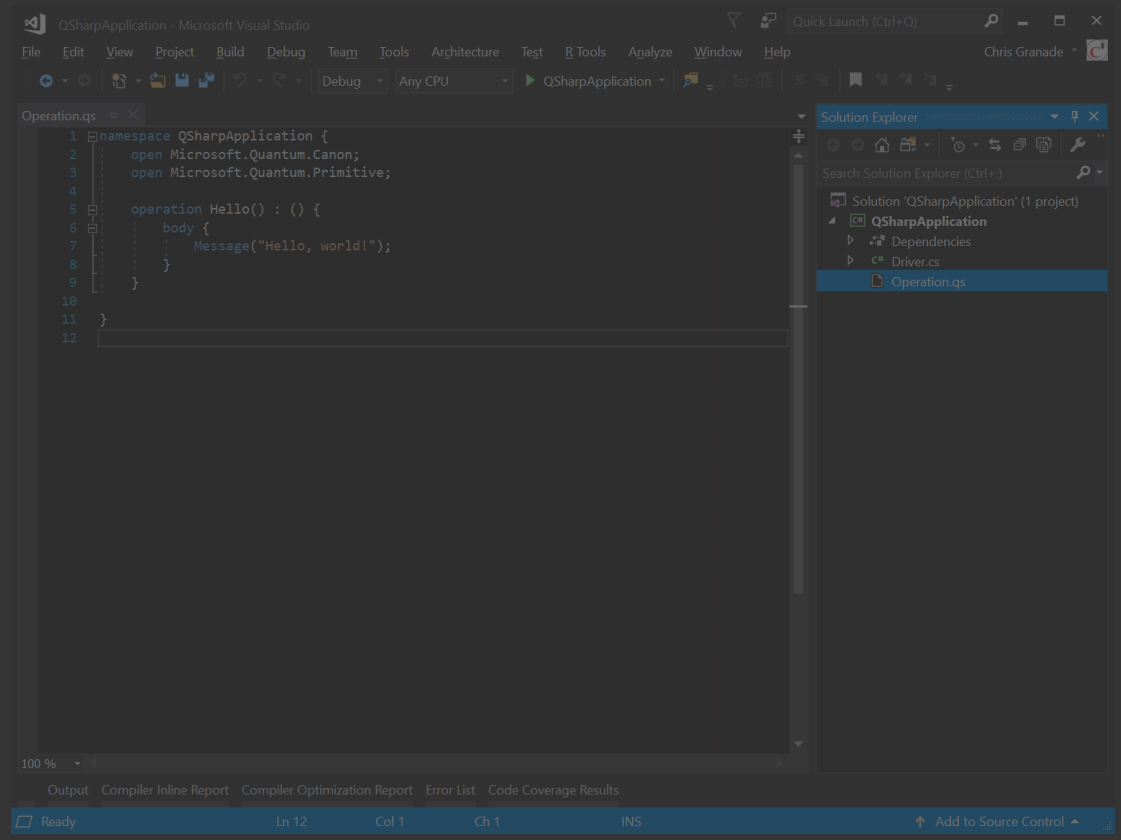
World-class C# and Q# debugging on Windows 10, unit testing, performance analysis.

Works with Q#, C#, F#, VB.NET, C++, Python, R, and more.

Visual Studio Code



Visual Studio 2017



I'll be showing Visual Studio Code today, but if you're using Windows 10 and would prefer to use Visual Studio 2017, great!

Full documentation for using Visual Studio 2017 is available at aka.ms/quantum-katas.

Getting Started with Q#

Q# programs are made up of *operations* and *functions*:

OPERATIONS

Quantum or nondeterministic classical code that can interact with and transform qubits.

FUNCTIONS

Purely deterministic classical code (e.g. `Sin`, `Cos`).

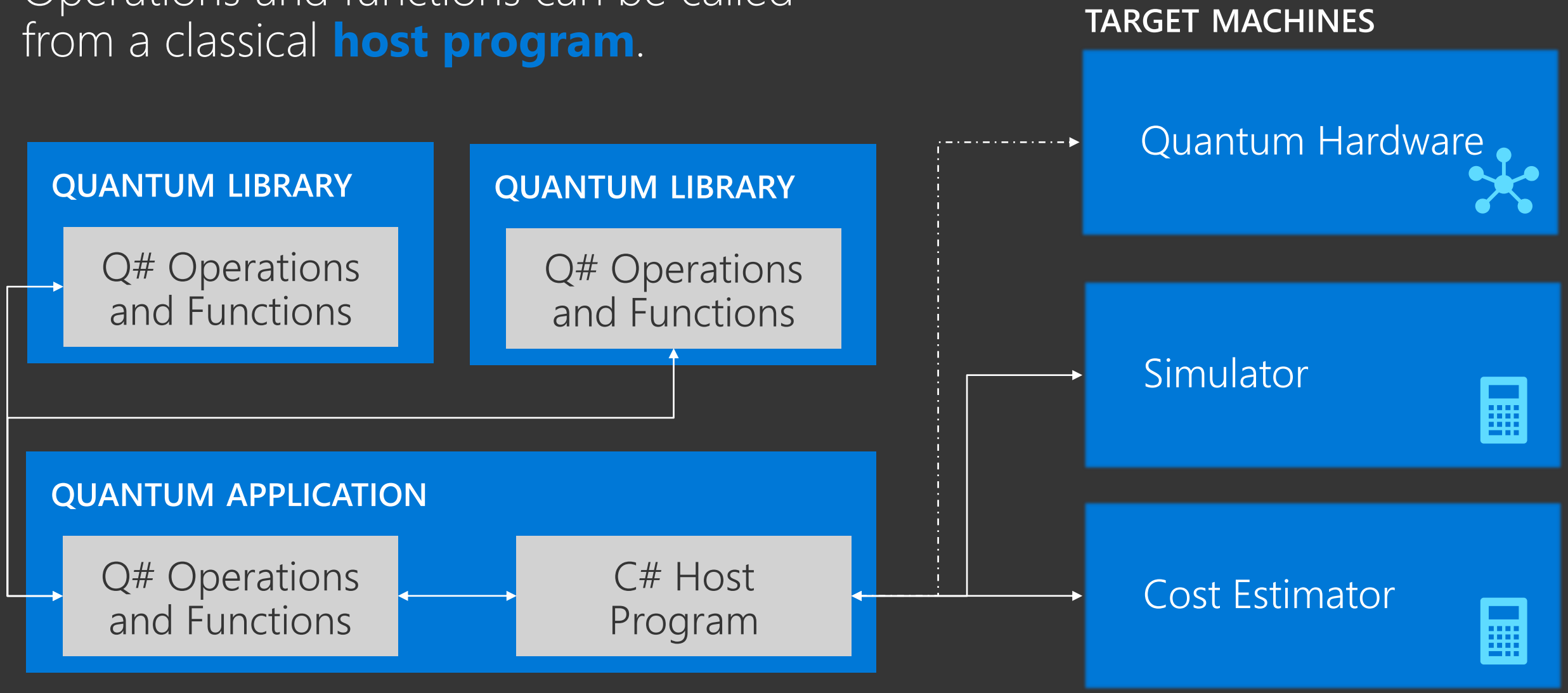
The Q# *prelude* includes built-in operations and functions:

- `X`, `Y`, `Z`
- `H`, `S`
- `R`, `R1`, `Rz`
- `M`, `Reset`
- `Message`

Complete documentation at:
docs.microsoft.com/quantum

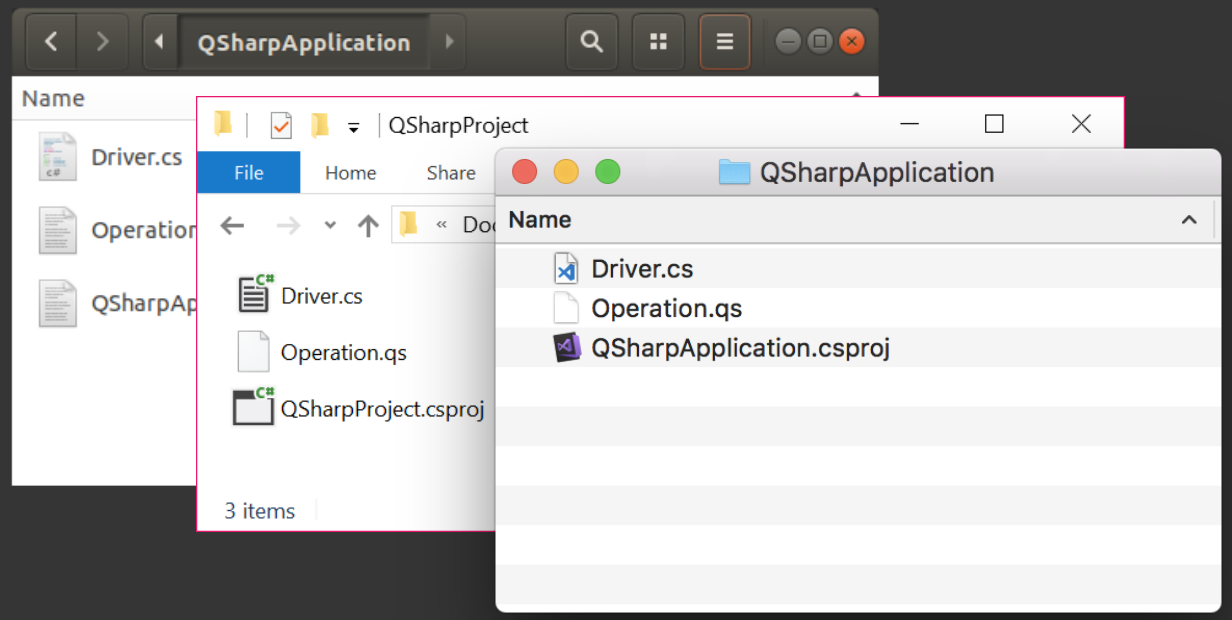
Q# Computation Model

Operations and functions can be called from a classical **host program**.



Working with .NET Core

Code in .NET Core is grouped into **projects**.



Each project compiles to a single cross-platform library or program.

QSharpApplication/

SOURCE CODE

QSharpApplication.csproj — project file
Driver.cs — classical (C#) source
Operations.qs — quantum (Q#) source

COMPILED CODE

bin/Debug/
    QSharpApplication.dll — compiled quantum app
obj/

Working with .NET Core

Code in .NET Core is grouped into **projects**.

Each project compiles to a single cross-platform library or program.

QSharpApplication/

SOURCE CODE

QSharpApplication.csproj — project file
Driver.cs — classical (C#) source
Operations.qs — quantum (Q#) source

COMPILED CODE

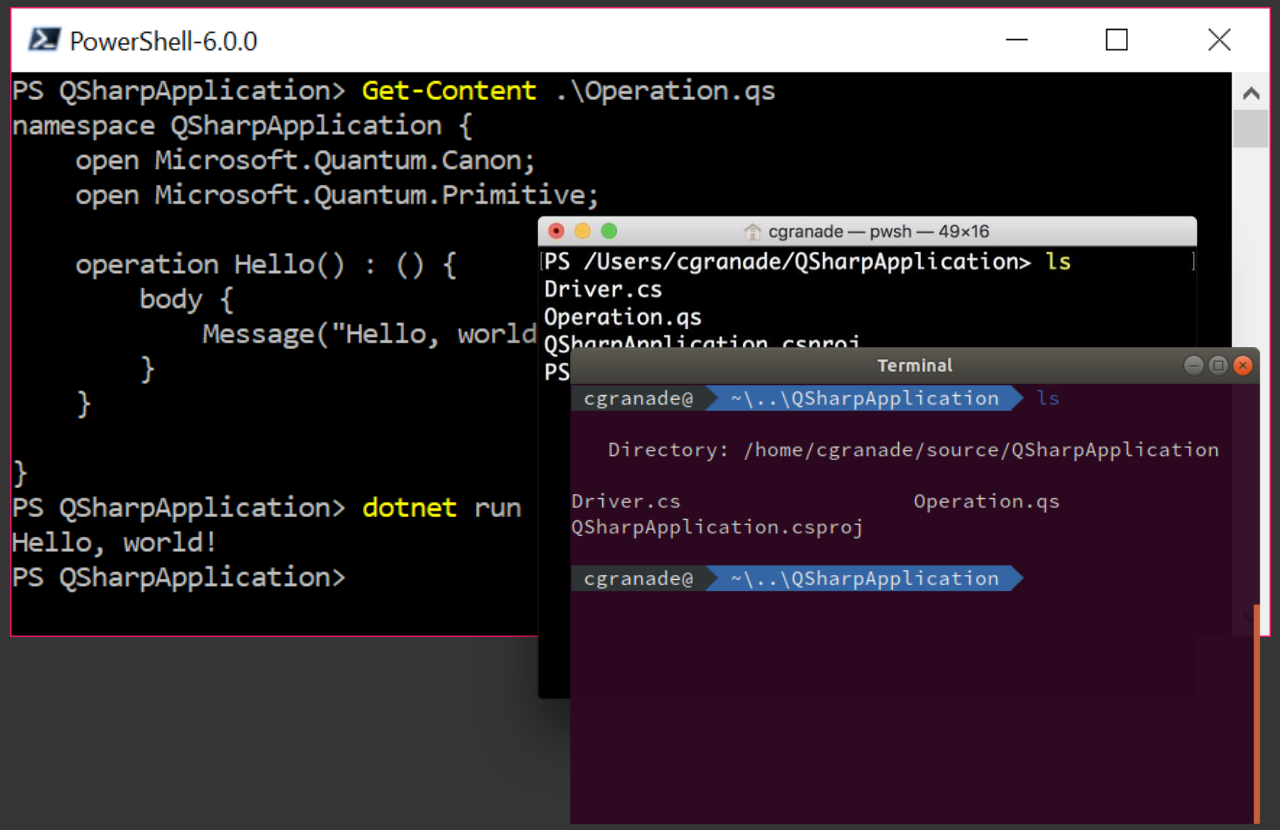
bin/Debug/
    QSharpApplication.dll — compiled quantum app
obj/

You may also see a *solution file* (`*.sln`).

This collects one or more related projects for loading in Visual Studio 2017.

Working with .NET Core

The **dotnet** tool provides a rich command line interface into .NET Core.



`dotnet help`
`dotnet *cmd* --help`

`dotnet new`
Makes a new project.

`dotnet build`
Downloads all the packages needed by a project and runs the compiler.

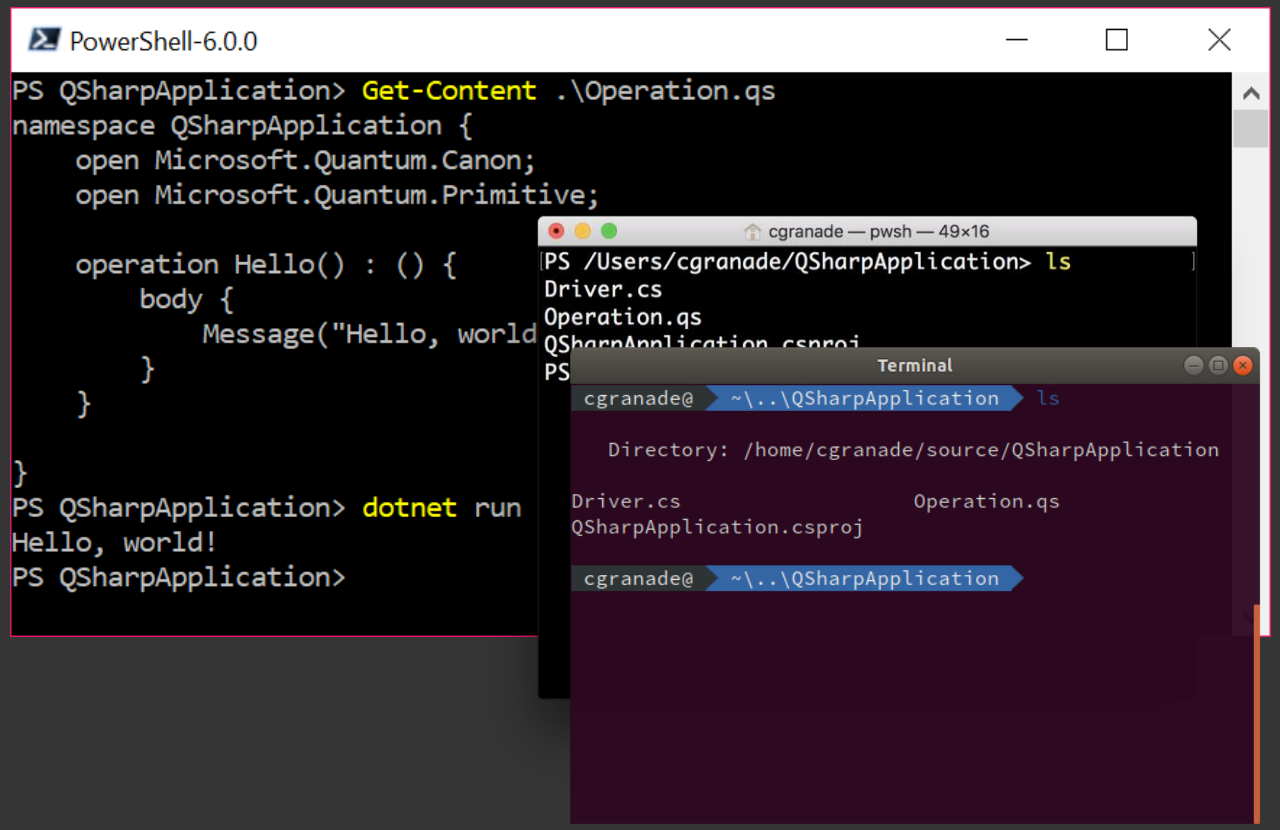
`dotnet run`
Runs the project, rebuilding if necessary.

`dotnet test`
Runs any unit tests defined in a project.

`dotnet add package`
Installs a new package into the project.

Working with .NET Core

The **dotnet** tool provides a rich command line interface into .NET Core.



`dotnet help`
`dotnet cmd --help`

`dotnet new`
Makes a new project.

`dotnet build`
Downloads all the packages needed by a project and runs the compiler.

`dotnet run`
Runs the project, rebuilding if necessary.

`dotnet test`
Runs any unit tests defined in a project.

`dotnet add package`
Installs a new package into the project.

Q# By Example: A Classic Hello

Let's start by seeing how Q# and C# work together to make a complete program.

Types

`Int`
Signed 64-bit integers.

`Double`
Floating point numbers.

`Bool`
Either `true` or `false`.

`Result`
Either `Zero` or `One`.

`Pauli`
Either `PauliI`, `PauliX`, `PauliY`, or `PauliZ`.

`Range`
A sequence of integers.

`String`
Diagnostic message.

For any types T_0, T_1, T_2, …, T_n we can make a tuple:

$$(T_0,\ T_1,\ T_2,\ \ldots,\ T_n)$$

We can also make *arrays* of any type T, denoted T`[]`.

```
(Result[], (Pauli, Pauli[]))
    (Int, Bool)[]
      (Int[], Double)
```

A Bit About `Qubit`s

Q# allows us to interact with quantum data in a variety of ways:

- Prepare fresh qubits in $|0\rangle$.
- Transform quantum data with built-in operations.
- Measure qubits to get back classical data.

Example

We can get random classical bits by:

- Starting with a fresh qubit.
- Using the **H** operation to prepare superposition.
- Measuring and collapsing the qubit to either $|0\rangle$ or $|1\rangle$.

Representing States and Operations

Qubit states can be represented as vectors,

$$|0\rangle \coloneqq \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad |1\rangle \coloneqq \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Operations then transform qubit states by matrix multiplication.

E.g.:

$$H \coloneqq \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} / \sqrt{2}, \qquad T \coloneqq \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}.$$

CONCEPTUAL INTRODUCTION AT

aka.ms/quantum-docs/the-qubit

Q# By Example: QRNGs

Let's make things quantum by generating some random numbers using the $|+\rangle \coloneqq (|0\rangle + |1\rangle) / \sqrt{2}$ state.

Example: Qrng

```
operation NextRandomBit() : Result {
  body {
    mutable result = Zero;
    using (qubits = Qubit[1]) {        // Prepare in |0⟩.
      H(qubits[0]);                    // H|0⟩ = |+⟩.
      set result = M(qubits[0]);       // Measure ⟨0|.
      Reset(qubits[0]);                // Reset to |0⟩.
    }
    return result;                     // Return to
  }                                    // classical host.
}
```

Example: Qrng

```
operation NextRandomBit() : Result {
  body {
    mutable result = Zero;
    using (qubits = Qubit[1]) {
      H(qubits[0]);
      set result = M(qubits[0]);
      Reset(qubits[0]);
    }
    return result;
  }
}
```

Let's look closer at what's going on when we call the `H` operation.

```
// Prepare in |0⟩.
// H|0⟩ = |+⟩.
// Measure ⟨0|.
// Reset to |0⟩.

// Return to
// classical host.
```

States in Q#

The **Qrng** example points at two different ways of thinking about quantum operations:

INPUT *IS* A STATE

$|+\rangle = H\,|0\rangle$

INPUT *HAS* A STATE

```
H(qubit);
```

In Q#, we program with qubits, and let the target machine define how to represent states.

Each target machine might define states differently.

- Local simulator: state vector
- Trace simulator: tally counter
- Actual hardware: device ID

By working with qubits, code is reusable across all targets.

Heisenberg vs Schrödinger Programming

Problem:
How do we describe states when states aren't in Q#?

$$|+\rangle \coloneqq \frac{|0\rangle + |1\rangle}{\sqrt{2}} = H|0\rangle$$

$$|\beta_{00}\rangle \coloneqq \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \text{CNOT}\ H\ |00\rangle$$

Solution:
Think of operations as *programs* which prepare states when given $|0\rangle$.

Operations are *first-class* in Q#, so we can pass them as values.
(More on this later.)

Allocating and Using Qubits

```
operation NextRandomBit() : Result {
  body {
    mutable result = Zero;
    using (qubits = Qubit[1]) {          // Prepare in |0⟩.
      H(qubits[0]);                      // H|0⟩ = |+⟩.
      set result = M(qubits[0]);         // Measure ⟨0|.
      Reset(qubits[0]);                  // Reset to |0⟩.
    }
    return result;                       // Return to
  }                                      // classical host.
}
```

This works because we agree to always prepare qubits in the same state, labeled |0⟩.

Representing Multiple Qubits

For multiple qubits, state vectors have an entry for every bitstring,

$$|00\rangle \coloneqq \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |01\rangle \coloneqq \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |10\rangle \coloneqq \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |11\rangle \coloneqq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Operations can be written as matrices on these states. E.g.,

$$\mathrm{CNOT} \coloneqq \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

CONCEPTUAL INTRODUCTION AT

aka.ms/quantum-docs/multiple-qubits

Q# By Example: Entanglement

We now have everything we need to make and measure pairs of entangled qubits.

Q# By Example: Teleport I

Q# can help us *think* conceptually about algorithms.

What is a program?

```
#include <stdio.h>


main( )
{
        printf("hello, world\n");
}
```

```
$ clang hello.c
$ ./a.out
hello, world
```

What is a program?

- A *description* of how to implement a computational task.
- *Interpreted* by a compiler to produce a lower-level program.
- *Interpreted* by a processor to run the program.

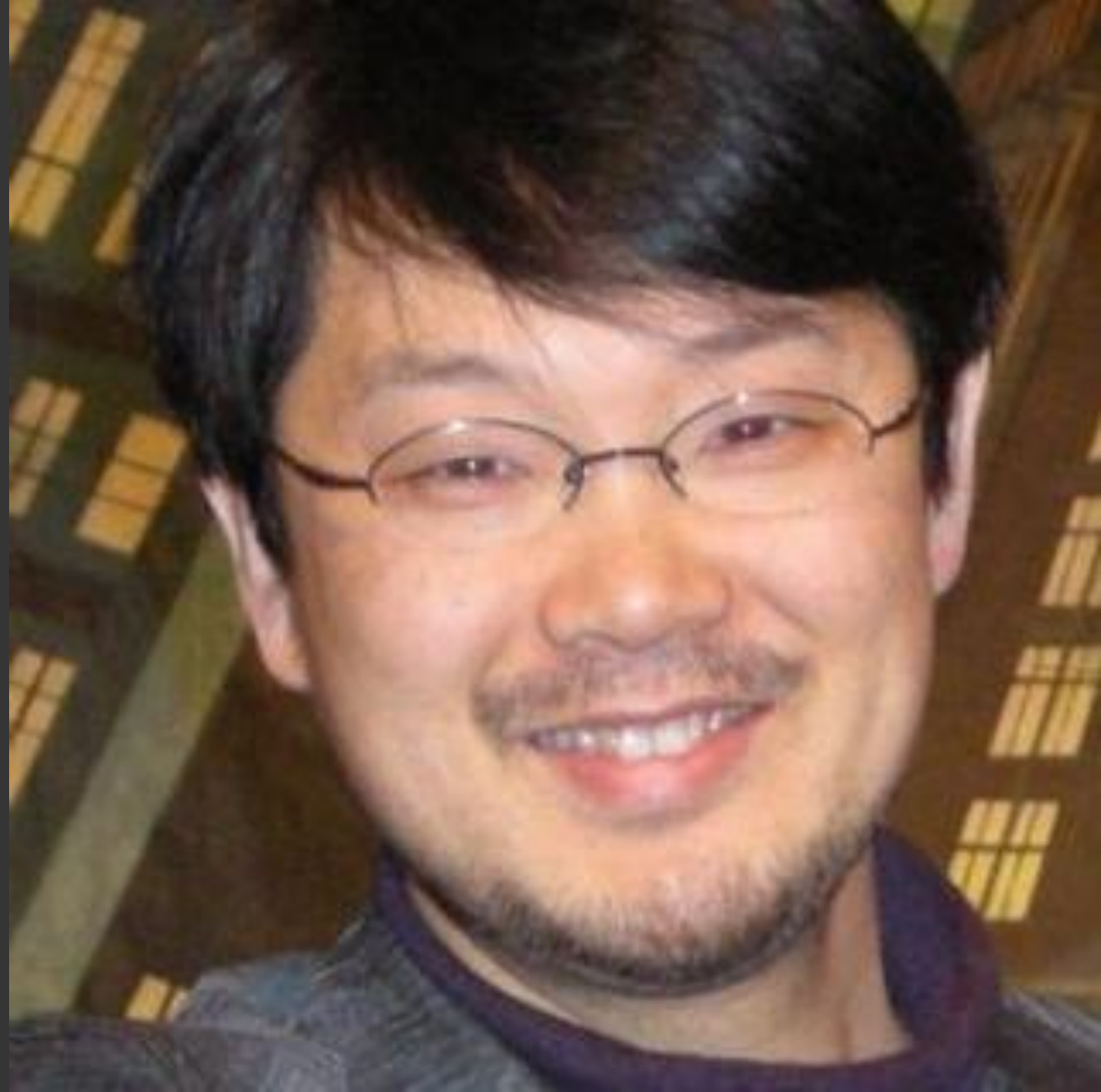
Description and interpretation are inherently about *communication* — hence the need for *languages*.

"We dissect nature along lines laid down by our native language."

—Benjamin Lee Whorf, linguist

"If programmers think in programming languages, they must influence thoughts as much as natural languages do."

—Yukihiro Matsumoto, lead designer of Ruby



Example: Four Views of Iteration

```
for (int i = 0; i < n; i++) {
    /* … */
}
```

```
for element in array: # …
```

```
map(fn, array)
```

```
MOV cx,10
loop:
  ; …
  dec cx
  jnz loop
```

Each expresses same idea, but communicates a different focus:

e.g.: architecture-specific implementation, data structures, or composition.

We dissect **quantum** algorithms along lines laid down by our programming languages.

Q# can help us **think** conceptually about algorithms.

Case Study: Conjugating With Unitaries

Common quantum algorithm concept:

UVU^{+}

We want to make it easier to think in terms of this concept:

```
let PhaseIncrement = With(QFT, Increment);
```

Let's explore the features we need to support conceptual thinking.

Functors

```
// U+ is an example of the adjoint functor.
operation EPR(left : Qubit, right : Qubit) : () {
    body {
        H(left);
        (Controlled X)([left], right);
    }
    adjoint auto;
}

// Can undo the operation with (Adjoint EPR).
```

Example: With

```
operation WithCA(
    outer : (Qubit[] => () : Adjoint),
    inner : (Qubit[] => () : Adjoint, Controlled),
    target : Qubit[]
) : () {
    body {
        outer(target);                  // U
        inner(target);                  // V
        (Adjoint outer)(target); // U+
    }
    adjoint auto; controlled auto;
    controlled adjoint auto;
}
```

Communicates the algorithmic pattern UVU^{+}.

E.g.:

```
let PhaseIncrement
    = With(QFT, Increment);
```

Case Study: Conjugating With Unitaries

We now have a language for clearly expressing the UVU^{+} concept.

Let's go further by reducing the *cognitive load* imposed by that expression.

e.g.: Currently, a user has to think about the type `Qubit[]` to use `With`.

Type-Parameterized Functions and Operations

```
// Using type parameters, we can express
// that a concept applies across different types.
function Fst<'T, 'U>(pair : ('T, 'U)) : 'T {
    let (first, second) = pair;
    return first;
}

let pair = (3, "Hello"); // pair : (Int, String)
Fst(pair);               // Returns 3.
```

Function Types

Let `'T` and `'U` be any two types.

Then `'T -> 'U` is a function that takes `'T` as an input and returns `'U`.

SINGLETON–TUPLE EQUIVALENCE

`'T` and `('T)` are exactly the same type for all `'T`.
Thus, all functions take and return a single tuple.

EXAMPLE

```
function Map<'T, 'U>(fn : 'T -> 'U, in : 'T[]) : 'U[] {
    let n = Length(arr);
    mutable out = new 'U[n];
    for (idx in 0..n - 1) { set out[idx] = fn(in[idx]); }
    return out;
}
```

Operation Types

Similarly, `'T => 'U` is an operation that takes `'T` as an input and returns `'U`.

Adjoint and Controlled can be specified as well.

`('T => () : Adjoint, Controlled)`

EXAMPLE

```
operation ApplyToEach<'T>(op : ('T => ()), targets : 'T[]) : () {
    body {
        for (idx in 0..Length(targets) - 1) { op(targets[idx]); }
    }
}
ApplyToEach(H, register);
```

Passing States as Operations

```
operation QuantumSearch(nIterations : Int, markedQubit : Qubit,
databaseRegister: Qubit[]) : () {
  body {
    StatePreparationOracle(markedQubit, databaseRegister);
    // Loop over Grover iterates.
    for (idx in 0..nIterations - 1) {
      ReflectMarked(markedQubit);
      ReflectStart(markedQubit, databaseRegister);
    }
  }
}
```

We can't directly pass $|\psi\rangle$, but we can pass an operation O that prepares $|\psi\rangle$ when given qubits in $|0\rangle$.

FULL SAMPLE AT

aka.ms/quantum-samples/DatabaseSearch

Q# By Example: Teleport II

We can make teleport much more
reusable with first-class operations.

Functional Programming

```
function Add(a : Int, b : Int) : Int {
    return a + b;
}

let Sum = Fold(Add, 0, _);
Sum([3; 4; 5]); // Returns 12.

Map(Add, Zip([1; 2; 3], [10; 11; 12]));
// Returns [11; 13; 15].
```

The same features also make it easier to express some kinds of classical processing in terms of composition and combinations of functions.

Partial Application

```
operation PhaseEst(
    oracle : (Qubit[] => ()),
    prep   : (Qubit[] => ()),
    target : Qubit[]
) : Double { … }

PhaseEst(
    Exp(theta, [PauliZ], _),
    ApplyToEach(H, _),
    target
);
```

We can make new functions and operations by specifying a subset of the inputs to a function or operation.

This lets us quickly combine existing callables to work well together.

Partial Application as Control Flow I

We can implement new functional constructs with partial application.

EXAMPLE

```
function ComposeImpl<'T, 'U, 'V>
         (outer : ('U -> 'V), inner : ('T -> 'U), target : 'T) : 'V
{ return outer(inner(target)); }

function Compose<'T, 'U, 'V>
         (outer : ('U -> 'V), inner : ('T -> 'U)): ('T -> 'V) {
    return ComposeImpl(outer, inner, _);
}
```

Partial Application as Control Flow II

Functions can also return partially applied operations, representing *classical* reasoning *about* the instructions given to a quantum device.

EXAMPLE

```
operation ApplyTwice<'T>(op : 'T => (), target : 'T) : () {
    body { op(target); op(target); }
}
function SquareOp<'T>(op : 'T => ()) : ('T => ()) {
    return ApplyTwice(op, _);
}
```

`SquareOp` has type `('T => ()) -> ('T => ())`.

Example: With

```
operation WithCA<'T>(
    outer : ('T => () : Adjoint),
    inner : ('T => () : Adjoint, Controlled),
    target : 'T
) : () {
    body {
        outer(target);                  // U
        inner(target);                  // V
        (Adjoint outer)(target); // U+
    }
    adjoint auto; controlled auto;
    controlled adjoint auto;
}
```

Can quickly express ideas like "conjugate the operation U by a Hadamard on each qubit."

```
let Conjugate = WithCA(
    ApplyToEach(H, _),
    _
);
Conjugate(U);
```

Q# By Example: Teleport III

We can quickly combine different operations together using partial application.

Diagnosing Quantum Programs

or How To Keep Moving When Everything Seems Broken

Quantum programming is still programming: mistakes and bugs still happen.

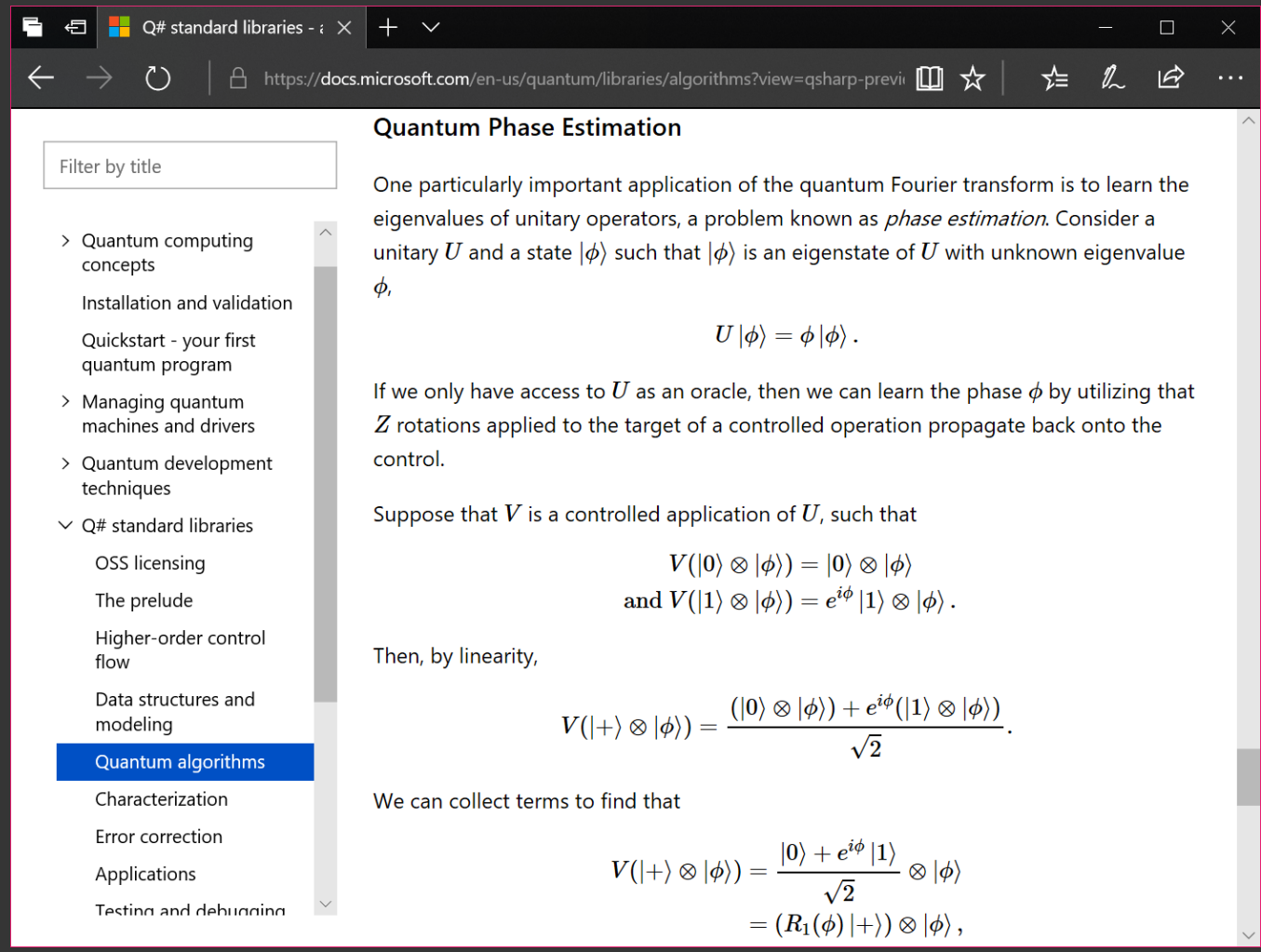
Let's look at a few ways to find and fix bugs!

Step 1

Read the Docs.

Many bugs come down to mismatched assumptions and conventions.

Checking back with the docs can help.



OPERATION AND FUNCTION REFERENCE

aka.ms/qsharp-ref

"printf" Debugging

`Message : String -> ()` emits diagnostics to C# driver.

By default, messages are printed to the console.

Especially useful for finding classical mistakes.

INTERPOLATED STRINGS

`$"{expr}"` inserts diagnostic information about `expr` into a `String` for use with `Message`.

EXAMPLE

```
function Hello() : () {
    let x = 42;
    Message($"Hello, {x}."); // Prints “Hello, 42.” to the console.
}
```

Assertions About States

Simulators need not obey the No-Cloning Theorem.

The `Assert*` operations executed on a simulator kill a Q# program if it would produce the "wrong" measurement.

ANTHROPIC PRINCIPLE

Observed effects must be compatible with their being an observer.

EXAMPLE

```
u (msg);
TeleportMessage (msg, there);
(Adjoint u)(there);

AssertQubit(Zero, there);
```

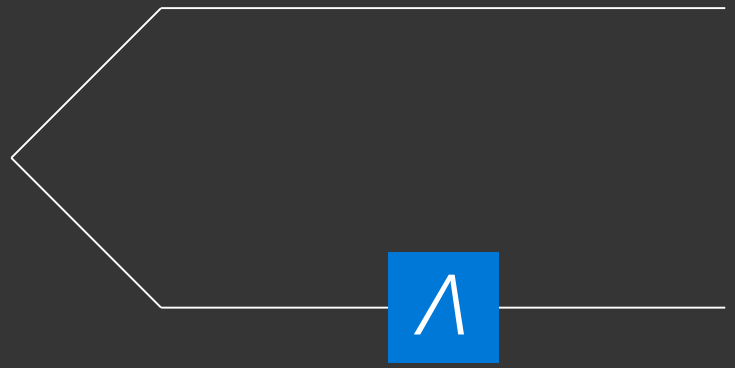
We used `AssertQubit` earlier to test the validity of our teleport operation.

The Choi–Jamiłkowski Isomorphism

Quantum processes are isomorphic to quantum states,

$$|i\rangle\langle j| \leftrightarrow |i\rangle|j\rangle.$$

Define $J(\Lambda)$ by acting Λ on half of an entangled pair.



EXAMPLE

```
operation IdentityTeleport(q : Qubit[]) : () {
    body { using (aux = Qubit[1]) {
        Teleport(q[0], aux[0]);
        SWAP(q[0], aux[0]);
    } }
}

operation TeleportationTest() : () {
    body {
        // Process assertions are a
        // special case of state assertions.
        AssertOperationsEqualReferenced(
            IdentityTeleport, NoOp, 1);
    }
}
```

FULL SAMPLE AT

aka.ms/quantum-samples/UnitTesting

Dumping Diagnostic Information

`DumpMachine<'T> : 'T -> ()` instructs target machines to report their diagnostic information.

> The only observable effect of functions is their return value.
>
> `DumpMachine` and `Message` return `()`, which has only one valid value `()`.
>
> This allows for calls to be stripped with no observable consequences.

```
DumpRegister<'T> :
    ('T, Qubit[]) -> ()
```

reports diagnostic information about a subset of qubits.

Information is target machine–dependent:

- `QuantumSimulator` reports state vectors.
- `QCTraceSimulator` does not report any additional info.
- Future machines may provide different information entirely (e.g.: internal IDs for `Qubits`).

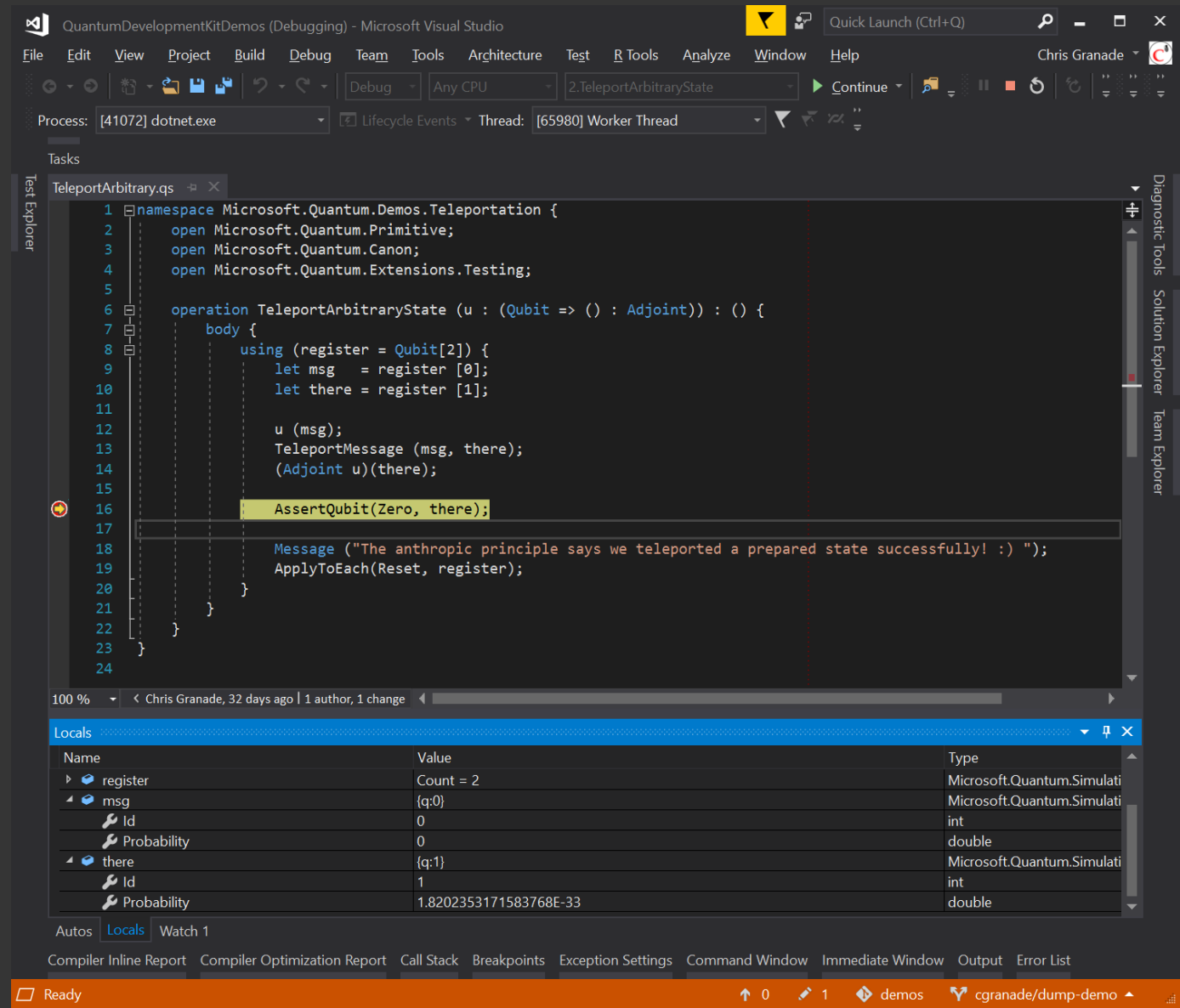
Q# By Example: Diagnostics

Let's see `DumpMachine` in action.

Step-By-Step Debugging in Visual Studio 2017

Visual Studio 2017 goes further, using diagnostic information exposed by target machines to help find bugs.

- Can step between C# and Q# code.
- Locals shows the probability of obtaining **`Zero`** for hypothetical measurements.
- Debugger and Test Explorer integrate to help fix unit tests.



The Quantum Katas

Let's get hands-on!

Each kata has a **`Tasks.qs`** file with blanks to fill in with your code.

Run **`dotnet test`** to check your answers.

RUNNING THE QUANTUM KATAS

1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS

aka.ms/quantum-katas

The Quantum Katas

HINTS

- If you get stuck in Visual Studio Code, search the Command Palette.
  Ctrl + Shift + P or ⌘ + Shift + P
- `ReferenceImplementation.qs` has the solutions — don't look unless you get *really* stuck!
- Your quick references have a lot of useful information, including function and operation summaries and links to documentation.

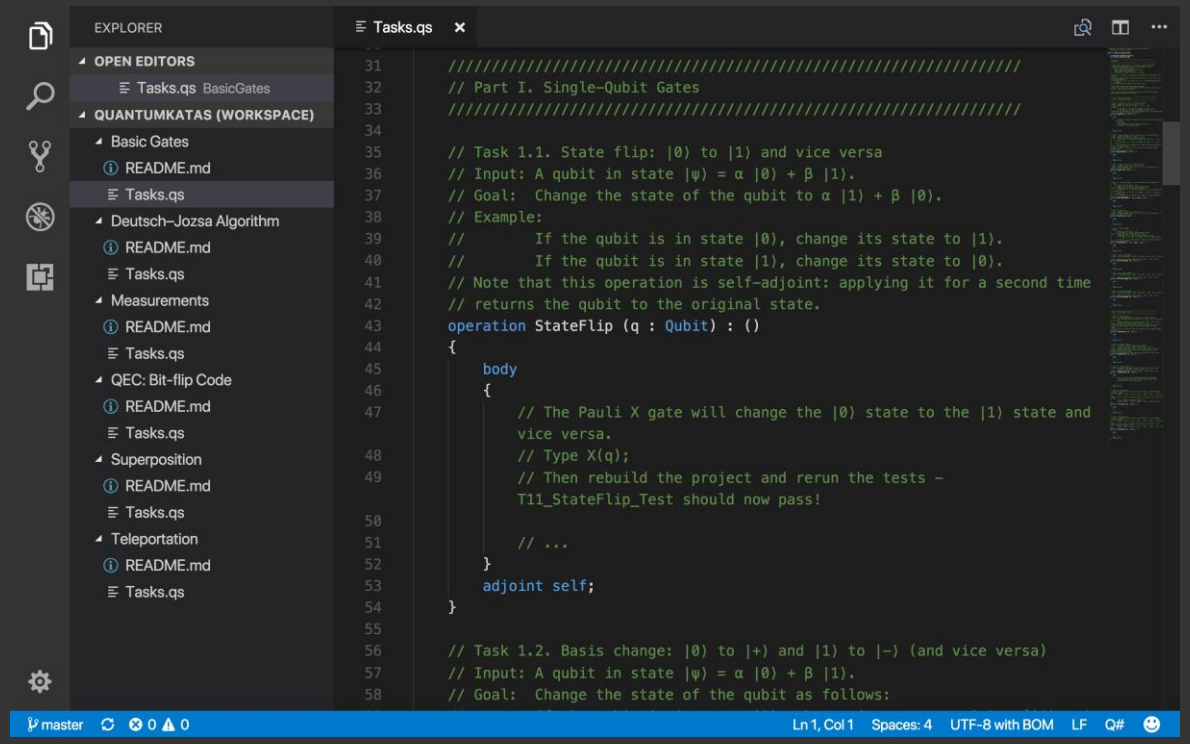
RUNNING THE QUANTUM KATAS

1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS
aka.ms/quantum-katas

The Quantum Katas



RUNNING THE QUANTUM KATAS

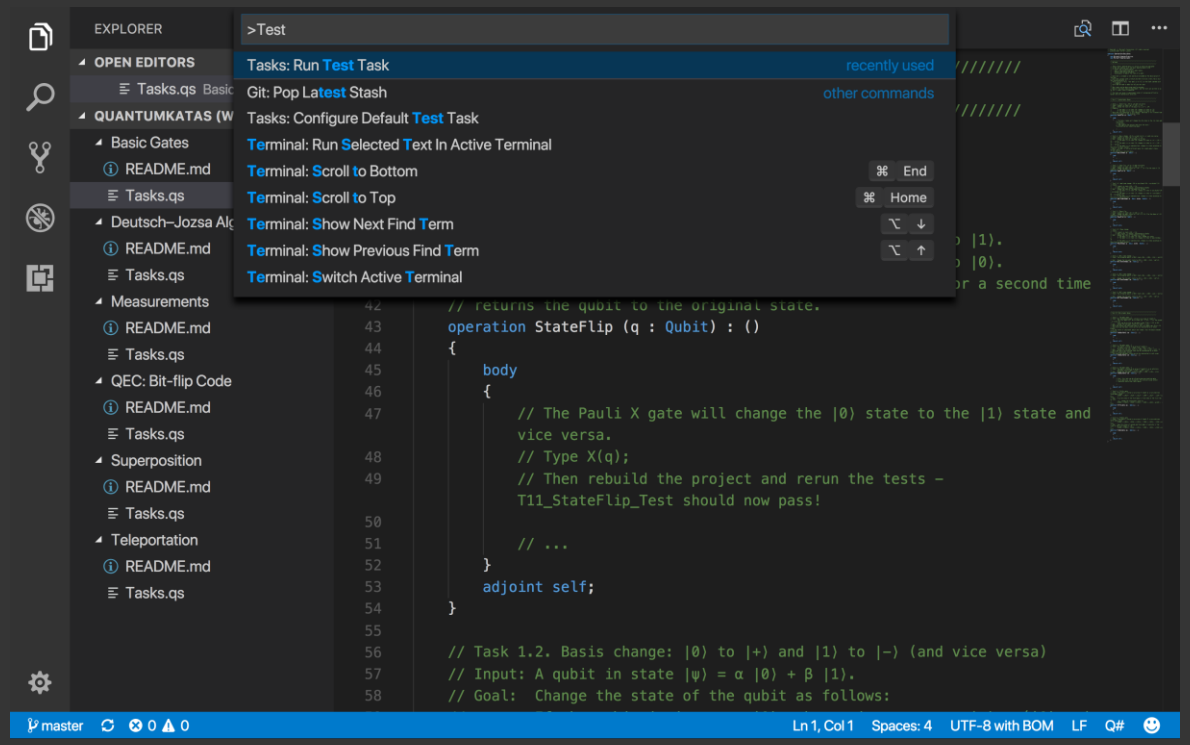
1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS

aka.ms/quantum-katas

The Quantum Katas



RUNNING THE QUANTUM KATAS

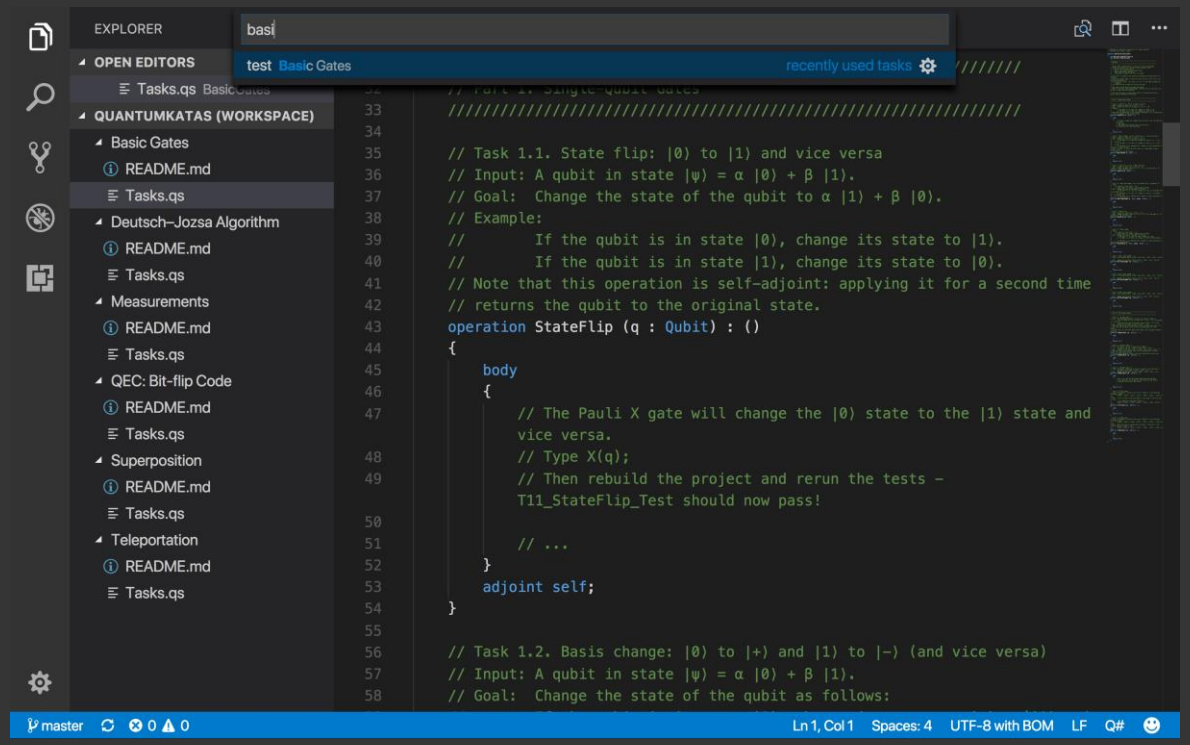
1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS

aka.ms/quantum-katas

The Quantum Katas



RUNNING THE QUANTUM KATAS

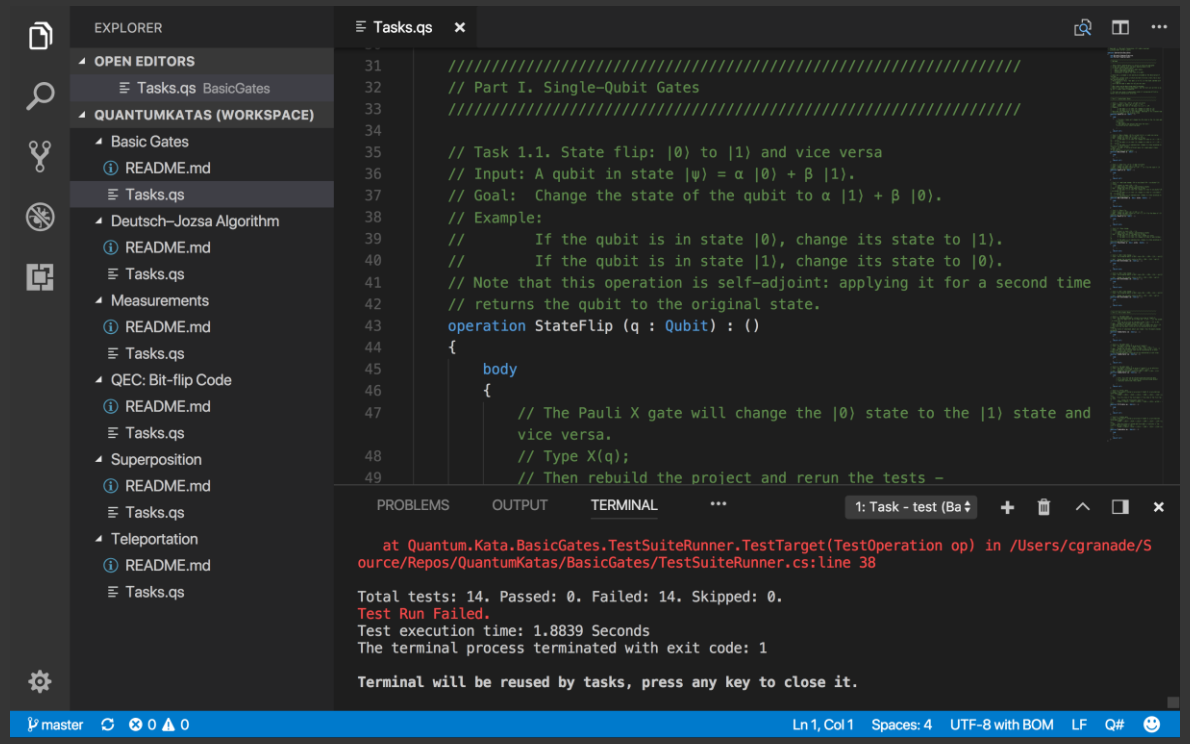
1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS

aka.ms/quantum-katas

The Quantum Katas



RUNNING THE QUANTUM KATAS

1. Download the Quantum Katas
2. Open `QuantumKatas.vscode-workspace`.
3. Run **Tasks: Run Test Task** from the Command Palette.

Six katas to choose from — pick one and have fun!

KATA DOWNLOAD AND INSTRUCTIONS

aka.ms/quantum-katas

The Quantum Development Kit Community

Many ways to get involved!

Feedback
quantum.uservoice.com

Documentation
github.com/MicrosoftDocs/quantum-docs-pr

Open Source Contributions
github.com/Microsoft/Quantum

Social Media
#qsharp • @MsftQuantum