Type	Description
`complex`	This data type represents a complex number. We can build this type ~~is~~ from either two `int`s or two `float`s of equal size, depending on the precision you want. This means we pass either an `int` or a `float` as the size, e.g. `complex[int[32]] name;`. The keyword `im` represents the complex unit $i = \sqrt{-1}$.
`bool`	Like the `bit`, this type can take one of two values, in this case `true` or `false`.

Constants

We can declare constant (immutable) classical types using the `const` keyword.

```
const uint[16] x = 44;
```

Shorthands

There are some shorthands useful when declaring classical types.

- When declaring an array of bits, we can use a string of `0` and `1` characters~~. E.g.~~ `bit[10] my_bits = '0011111011\';`.
- We can use scientific notation to declare large or small numbers~~. E.g.~~ `float[32] name = 2.34e5;`.
- QASM supports some popular mathematical constants that we can use to declare types~~. For example:~~
- When declaring `int`s and `float`s, we can use th[illegible]ywords `pi`, `tau`, and `euler`~~. E.g.~~ `float[32] name = 4*euler;`.
- When declaring a `complex` type, we can use the `im` keyword to represent the imaginary unit~~. E.g.~~ `complex[float[32]] my_complex = 3.1 + 1.2im;`.

Arrays of Classical Types

We've already seen how to create arrays of `bits` and `qubits` with the square bracket syntax:

```
qubit[3] qr;  // array of 3 qubits
bit[3] cr;    // array of 3 bits
```

But we can also create arrays of the higher-level classical types through the `array` keyword. This keyword should be followed by the type~~,~~ and the size of the array in square brackets. For example, the code ~~below~~ creates an array of 10 16-bit `ints`, named `int_array`:

```
array[int[16], 10] int_array;
```

And we can access each element of `int_array` using square `brackets` as we did with `bit` and `qubit` arrays. We can also specify all the values of an array when declaring it using curly brackets.

```
array[float[64], 3] my_array = {0.1, 2.9, pi};
```

QASM also support multi-dimensional arrays, we just need to pass the extra dimension to the array constructor. In the code ~~below~~ we create a 2D array of 32-bit `units`, named `matrix`.

```
// 8x8, 2D array
array[uint[32], 8, 8] matrix;
```

Built-In Classical Instructions

QASM supports some common operations between classical data of the same type. All instructions shown here must have a data type on the left-hand side (LHS) of the instruction~~,~~ and a value of the same type on the right-hand side (RHS). For example, the assignment operator (=) sets the value of the data on the LHS to the value on the RHS.

```
bit x;  // declare bit
x = 1;  // set x to 1
```

All data types except `complex` support the comparison operators shown in Table 10-2 ~~below~~.

Table 10-2. Classical comparisons supported by QASM

Name	Symbol	Description
Equal to	`==`	Compares the value on the RHS to the value on the LHS~~,~~ and returns `True` if, and only if, both sides are equal
Not equal to	`!=`	Returns `False` if, and only if, both sides are equal
Less than	`<`	Returns `True` if LHS is less than RHS
Greater than	`>`	Returns `True` is LHS is greater than RHS
Less than or equal to	`<=`	Returns `True` if ~~the~~ LHS is less than or equal to ~~the~~ RHS
Greater than or equal to	`>=`	Returns `True` if ~~the~~ LHS is greater than or equal to ~~the~~ RHS

All numeric types (`int`, `float`, `angle`, and `complex`) support the basic arithmetic operations in Table 10-3 ~~below~~.

Table 10-3. Classical numeric instructions supported by QASM

Name	Symbol	Description
Addition	`+`	Returns value of LHS added to RHS
Multiplication	`*`	Returns value of LHS multiplied by RHS
Power	`**`	Returns value of LHS to the power (exponent) of RHS
Division	`/`	Returns value of LHS divided by RHS (note that dividing two `int`s returns an `int`)

Along with the operations in Table 10-3 ~~above~~, integers (`int`) also support the modulo operation shown in Table 10-4 ~~below~~.

Table 10-4. Classical integer instructions supported by QASM

Name	Symbol	Description
Modulo	`%`	Returns the value of the LHS modulo RHS (i.e. the remainder of LHS divided by RHS)

As with most programming languages, we can also condition operations on the state of a `bool` and the `if` keyword. The `if` keyword must be followed by the `bool` (in parentheses), then by the conditioned logic (in curly brackets). We can then follow this with the `else` keyword to trigger instructions only if the statement was `false`. For example, the code ~~below~~ calculate the maximum value of `a` and `b`~~,~~ and stores it in `c`.

```
if (a < b) {
    c = b;
} else {
    c = a;
};
```

With this classical logic, we can now encode more complicated quantum circuits, such as the famous quantum teleportation circuit shown in Figure 10-10.

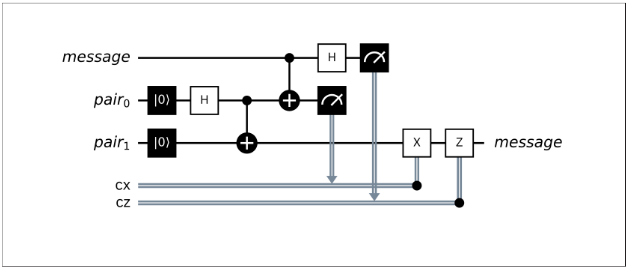

Figure 10-10. The quantum teleportation protocol

The QASM code ~~below~~ encodes the teleportation protocol ~~shown above.~~

```
// Declare two bits to be transmitted
bit cz;
bit cx;

// declare message and entangled pair
qubit message;
qubit[2] pair;
```

```
// Third party entangles pair
h pair[0];
cx pair[0], pair[1];

// Third party gives pair[0] to message sender
// and pair[1] to message recipient

// Message sender then entangles pair[0] with
// the message qubit, and measures
cx message, pair[0];
h message;
cz = measure message;
cx = measure pair[0];

// Message sender sends two classical bits to
// message recipient
if (cx == 1) {
    x pair[1];
}

if (cz == 1) {
    z pair[1];
}
```

QASM supports `while` loops, which repeat instructions as long as a condition is `true`. For example, the code ~~below~~ implements a very inefficient way of resetting a qubit (`q`) to $|0\rangle$.

```
qubit q;
bit b = 1;

while (b==1) {
    h q;
    b = measure q;
}
```

QASM also support `for` loops, which repeat instructions for each item in an array. For example, the code ~~below~~ applies a H-gate to each qubit in `qr`.

```
qubit[3] qr;
```

```
for q in qr {
    h q;
}
```

Building Quantum Programs

We have now covered everything needed to create rich, circuit-level quantum programs in QASM. In this section, we'll look at two QASM features that make managing and reusing programs easier.

Subroutines

Similar to custom gate definitions, we can combine both classical and quantum instructions into a *subroutine*. A subroutine definition starts with the `def` keyword, followed by the name of the subroutine. Next, we declare and name the data types the subroutine will act on (in parentheses), then indicate the data type that the subroutine returns (after a `->` symbol). The subroutine instructions follow, enclosed by curly brackets.

To illustrate, the code ~~below~~ creates a subroutine named `bell_measurement` that measures two qubits in the Bell basis. This subroutine takes an array of two `qubits` and returns two `bits`.

```
def bell_measure(qubit[2] qr) -> bit[2] {
    CX qr[0], qr[1];
    h qr[0];
    return measure qr;
}
```

To use this subroutine in a program, we follow the subroutine's name with the arguments, enclosed in parentheses.

```
qubit[2] qr;
bit[2] cr;

cr = bell_measure(qr);
```

Note that you can declare classical data types in the body of a subroutine~~,~~ but not quantum data types.

Inputs and Outputs

We've seen that instructions and subroutines can accept inputs and return outputs, and this is also true of entire QASM programs; when we declare classical data, we can prefix the declaration with the `input` keyword, which means the value of that data will only be known at run-time. In the code ~~below~~, we declare an array of 20 `angles` named `point`~~,~~ and tell QASM that this value will ~~only~~ be known at run-time.

```
    input angle[20] point;
```

By leaving some values unknown when compiling, we can avoid repeating long compilation and optimization processes. This is especially useful for near-term, variational algorithms that require many circuit runs with different gate parameters.

We can also specify any classical data types that our program should output using the `output` keyword. If we don't specify any outputs, then the QASM program will output *all* classical data.

Index

About the Authors

James L. Weaver is a developer, author, and speaker with a passion for quantum computing. He is a Java Champion, and a JavaOne Rockstar. James has written books including Inside Java, Beginning J2EE, the Pro JavaFX series, and Java with Raspberry Pi. As an IBM Quantum Developer Advocate, James speaks internationally about quantum computing with Qiskit at quantum and classical computing conferences.

~~Francis~~ Harkins

Colophon

The animal on the cover of *Qiskit Pocket Guide* is a rock quail (*Latin name*).

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *English Cyclopedia*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.