Classical functions

In [2]:

```
function HelloWorld() : Unit {
    Message("Hello, classical world!");
}
```

Out[2]:

- HelloWorld

In [41]:

```
%simulate HelloWorld
```

```
Hello, classical world!
```

Out[41]: ()

Magic commands

Magic commands are any command with the prefex `%`.

Eg. `%simulate` is a magic command.

Fun fact: magic commands are actually not a part of Q# but rather the Jupyter Notebook environment.

Quantum operations

In [5]:

```
operation MeasurePlusState() : Result {
    use qubit = Qubit();
    H(qubit);
    let result = M(qubit);
    Reset(qubit);
    return result;
}
```

Out[5]:

- MeasurePlusState

In [16]:

```
%simulate MeasurePlusState
```

Out[16]: One

Functions vs Operations

- **functions** are for classical logic only
  - will always return the same output given the same input

- **operations** are for quantum logic (and classical logic)
    - results may differ due to the nature of dealing with quantum states

Using vs borrowing

There are two ways to obtain qubits.

`use` qubit

Qubit is initialized to 0.

In [23]:

```
open Microsoft.Quantum.Measurement;

operation UseQubit(): Result {
    use q = Qubit();
    let result = MResetZ(q);
    return result;
}
```

Out[23]:

- UseQubit

In [31]:

```
%simulate UseQubit
```

Out[31]: Zero

`borrow` qubit

We do not know what state the qubit starts in.

In [32]:

```
operation BorrowQubit() : Result {
    borrow q = Qubit();
    let result = MResetZ(q);
    return result;
}
```

Out[32]:

- BorrowQubit

In [40]:

```
%simulate BorrowQubit
```

Out[40]: Zero

Structure of an `operation` in Q

In [46]:

```
operation EntangleTwoQubits(q1 : Qubit, q2 : Qubit) : Unit {

    H(q1);
    CNOT(q1, q2);
}
```

Out[46]:

- EntangleTwoQubits

In [48]:

```
operation MeasureEntangledPair() : (Result, Result) {

    //use multiple qubits
    use qubits = Qubit[2];

    EntangleTwoQubits(qubits[0], qubits[1]);

    let result1 = M(qubits[0]);
    let result2 = M(qubits[1]);

    // reset multiple qubits
    ResetAll(qubits);

    return (result1, result2);
}
```

Out[48]:

- MeasureEntangledPair

In [64]:

```
%simulate MeasureEntangledPair
```

Out[64]:

```
(Zero, Zero)
```

Q# Datatypes

Non-exhaustive list:

- int
- double
- bool
- qubit
- unit
- result

`let`, `set`, and `mutable`

We used

```
q#
let result = M(qubit);
```

in the previous example. Notice that if you use the `let` keyword, you can never bind that

In [71]:

```
operation LetKeyword() : Unit {
    let result = 2;
    set result = 3;
}
```

```
/snippet_.qs(3,9): error QS6303: An immutable identifier cannot be modified.
```

but what if we want a standard variable that we can re-assign? In that case we should use

```
mutable
```

In [70]:

```
operation MutableKeyword() : Unit {
    mutable x = 2;
    set x = 3;
}
```

Out[70]:

- MutableKeyword

Libraries

`Microsoft.Quantum.Measurement`

- `Reset()`
- `ResetAll()`
- `MResetX()`, `MResetY()`, and `MResetZ()`

`Microsoft.Quantum.Diagnostics`

- `DumpMachine()`

In [76]:

```
open Microsoft.Quantum.Diagnostics;

operation MeasureEntangledPair() : (Result, Result) {

    //use multiple qubits
    use qubits = Qubit[2];

    EntangleTwoQubits(qubits[0], qubits[1]);

    // Shows a graph of state probabilities!
    DumpMachine();

    let result1 = M(qubits[0]);
    let result2 = M(qubits[1]);

    // reset multiple qubits
    ResetAll(qubits);

    return (result1, result2);
}
```

Out[76]:

- MeasureEntangledPair

In [75]:

```
%simulate MeasureEntangledPair
```

Qubit IDs	0, 1		
Basis state (little endian)	**Amplitude**	**Meas. Pr.**	**Phase**
$\vert 0\rangle$	$0.7071 + 0.0000i$	50.0000%	↑
$\vert 1\rangle$	$0.0000 + 0.0000i$	0.0000%	↑
$\vert 2\rangle$	$0.0000 + 0.0000i$	0.0000%	↑
$\vert 3\rangle$	$0.7071 + 0.0000i$	50.0000%	↑

Out[75]: `(One, One)`

Standard gates in Q

- `X(q);`
- `Y(q);`
- `Z(q);`
- `H(q);`
- `CNOT(q);`

In [84]:

```
operation Test(q1: Qubit, q2: Qubit) : Unit {
    H(q1);
    S(q1);
    T(q1);
    CNOT(q1, q2);
    X(q1);
    Y(q1);
    Z(q1);
}
```

Out[84]:

- Test

Loops in Q

In [86]:

```
operation WalshHadamardTransform() : Unit {

    let n = 10;
    use qubits = Qubit[n];

    for qubit in qubits {
        H(qubit);
    }

    mutable results = new (Int, Result)[0];
    for index in 0 .. Length(qubits) - 1 {
        set results += [(index, M(qubits[index]))];
    }
}
```

Out[86]:

- WalshHadamardTransform

Repeat until success loops

In [87]:

```
repeat {
    //...
}
until condition;
```

```
/snippet_.qs(1,1): error QS4004: Statements can only occur within a callable
or specialization declaration.
```

ApplyToEach

In [88]:

```
using (register = Qubit[3]) {
    ApplyToEach(H, register);
}
```

```
/snippet_.qs(1,1): error QS4004: Statements can only occur within a callable
```

```
or specialization declaration.
```

Arrays in Q

Arrays can be a bit of work in Q#.

In [ ]:
```
mutable result = new Int[n];

for index in 0 .. n-1 {
    if M(qubit[index]) == Zero {
        set result w/= index <- 0;
    } else {
        set result w/= index <- 1;
    }
}
```

`w/` operator

`w/` is a ternary operator (ie. it takes 3 arguments).

- `arg1` : old array
- `arg2` : index
- `arg3` : new element

All in action:

```
mutable a = [2, 4, 6];
set a = a w/ 0 <- 1;
```

So now, our array is:

```
a = [1, 4, 6]
```

But we can shortcut having to write `a` twice by using `w/=` :

```
mutable a = [2, 4, 6];
set a w/= 0 <- 1;
```

Functors

There are two:

- `adj`
- `ctl`

In [3]:

```
operation Adder(pos: Qubit[]) : Unit is Ctl+Adj {
    CNOT(pos[0], pos[1]);
    X(pos[0]);
}
```

Out[3]:

- Adder