Note: this is currently WIP and not finalised

TODO: reorganize the sections in this

QScript Syntax

Comments

Comments can be added using the # character or `//`. Anything following a # or `//` is ignored by compiler as a comment.

For example:

```
# a comment
// another comment
fn main(){ # This is a comment
    # This also is a comment
}
```

Multi line comments can be written by enclosing them in between `/*` and `*/` like:

```
/*
this
is
a
multiline
comment
*/
```

Loading external scripts/libraries

The `load` keyword can be used to create an instance of an existing script. A script is similar to a struct:

```
debug.qscript:
    var string prefix = "debug: ";
    pub var auto log = (string msg) -> { ... };

main.qscript:
    var auto logger = load debug;
    pub var auto this = () -> {
        logger.log("started");
    };
```

is equivalent to:

```
main.qscript:
    struct SomeInaccessibleType{
        var string prefix = "debug: ";
        pub var auto log = (string msg) -> { ... };
    }

    var SomeInaccessibleType logger;
    fn this(){
        logger.log("started");
    }
```

In the first case, the `load` debug statement creates an instance of the debug`.qscript` script, and returns it.

assigning a "namespace" of sorts as shown above is not necessary, a library can be loaded so that its symbols become accessible as global symbols:

```
load debug;
```

This will create an instance of debug and merge it's symbols with current script.

Similarly, a script can be loaded and all it's public symbols made public:

```
pub load debug;
```

Visibility

All script members are by default private, and only accessible inside the script.

The pub keyword can be prefixed to make a member public:

```
fn somePrivateFunction(int arg){
    # body of private function
}
pub struct SomeStruct{
    var int someInt, someOtherInt; # these data members are private
    pub var int x; # this is public
}
```

Functions

QScript has first class functions. Functions can be assigned to a variable (of appropriate type), or passed as an argument, they behave as normal variables.

Function Data Type

To create a variable that can store a function:

```
var fn RETURN_TYPE(arg0_type, arg1_type, ..) someFunc;
// or
var auto someFunc = someExistingFunction; // assignmment necessary here
```

following this, the function can be called using the `(..)` operator:

```
someFunc(arg0, arg1);
```

The Function Data Type is a reference data type, annotating it with a `ref` is not necessary, and not allowed.

Function Definition

```
[pub] fn [ref] [RETURN_TYPE] FUNCTION_NAME(
        [ref] arg0_type arg0,
        [ref] arg1_type arg1){
    # function body
}
// or
[pub] var auto FUNCTION_NAME = (
        [ref] arg0_type arg0,
        [ref] arg1_type arg1) -> [RETURN_TYPE]{
    # function body
};
```

- pub (optional) will make the function public
- `ref` (optional) will make it so parameters/return is passed by reference not value
- `RETURN_TYPE` (optional) is the return type of this function.
- `FUNCTION_NAME` is the name of the function
- `arg0_type` is the type for first argument
- `arg0` is the "name" for first argument
- more arguments can be added, and are to be separated by a comma.

Note that the `ref` part of return type is not a property of the function, but rather a part of the return data type.

A function without any arguments would be defined like:

```
fn [TYPE] FUNCTION_NAME(){
    # function body
}
# or
var auto FUNCTION_NAME = () -> TYPE{
    # function body
};
```

Anonymous Functions

```
fn int sumXTimes(int x, fn int(int) func){
    var int i = 0;
    var sum
    while (i < x)
        sum += func(i ++);
    return sum;
}
sumXTimes(5, (num) -> num * 5);
# is equivalent to:
sumXTimes(5, (int num) -> {return num * 5;});
# is equivalent to:
fn mul5(int num){ return num * 5; }
sumXTimes(5, mul5);
```

Returning From Functions

A return statement can be used to return a value from the function. The type of the return value, and the return type of the function must match. A return statement is written as following:

```
return RETURN_VALUE;
```

where RETURN_VALUE is the value to return. As soon as this statement is executed, the function execution quits, meaning that in the following code, writeln will not be called.

```
fn int someFunction(){
    return 0;
    writeln("this won't be written"); # because return terminates the execution
}
```

In case of functions that return nothing, `return;` can be used to terminate execution, like:

```
fn main(){
    # ..
    return;
    # anything below wont be executed
}
```

Function Calls

Function calls are made through the `()` operator like:

```
fnName(funcArg0, funcArg1, ...);
```

or in case of no arguments:

```
fnName();
```

The first argument to a function call can also be passed using the `.` operator:

```
fn int square(int i){
    return i * i;
}
fn main(){
    writeln(square(5)); # is same as following
    writeln(5.square); # writes 25
    # () operator not needed in this case
}
```

Member Functions

The `.` operator is also used to call member functions of a struct:

```
struct Position{
    pub var int x = 0, y = 0;
    pub fn flipXY(){
        var int temp = x;
        x = y;
        y = temp;
        writeln("scoped call");
    }
}
fn flipXY(ref Position pos){
    writeln("global called");
}

Position pos;
pos.flipXY(); # prints "scoped call"
```

`()` Overloading

```
struct SomeStruct{
    var int i;
}
fn opFnCall(SomeStruct strct, int a){
    writeln("SomeStruct() called with a = " ~ toString(a));
}
fn main(){
    var SomeStruct s;
    s(5); # prints: SomeStruct() called with a = 5
}
```

Overloading `()` operator for `function` types is not allowed, as that would break a lot of things.

See Operator Overloading section for more details

Data Types

QScript has these basic data types:

- `int` - a signed 32 or 64 bit integer (`ptrdiff_t` in DLang is used for this)
- `double` - a floating point (`double` in DLang)
- `char` - an 8 bit character
- `bool` - a `true` or `false` (Dlang `bool`)
- `auto` - in above examples, `auto` can be substituted for `X` in cases where inline assignment occers and compiler is able to detect intended type.
- `ref X` - reference to any of the above (behaves the same as `X` alone would)
- `fn ...(...)` - a Function Data Type (see above)

`int`

This can be written as series (or just one) digit(s).

Can be written as:

- Binary - `0B1100` or `0b1100` - for 12
- Hexadecimal - `0xFF` or `0XfF` - for 15
- Series of `[0-9]+` digits

`int` is initialised as `0`.

`double`

Digits with a single `.` between them is read as a double. `5` is an int, but `5.0` is a double.

`double`s are initialised as `0.0`

`char`

This is written by enclosing a single character within a pair of apostrophes:

`'c'` for example, or `'\t'` tab character.

initialised as ascii code `0`

`bool`

A `true` (1) or a `false` (0).

While casting from `int`, a non zero value is read as a `true`, zero as `false`.

initialised as `false`

`ref` references

These are aliases to actual variables. A `ref` must always be initialised to reference some variable:

```
int i = 0;
var ref int r = i; # r is now a reference of i
# r can now be used as if it's an int.
r = 1;
writeln(r); # 1
writeln(i); # 1
```

after initialising, a ref cannot be made to reference another variable.

auto variables

The `auto` keyword can be combined with others in the following order:

```
var [ref] auto
```

This is necessary since the `auto` detection will not pick up whether it should be made `ref`

Structs

These can be used to store multiple values of varying or same data types.

They are defined like:

```
struct STRUCT_NAME{
    [pub] var DATA_TYPE1 NAME1;
    [pub] var DATA_TYPE2 NAME2;
    [pub] var auto print1 = () -> writeln(NAME1);
    [pub] fn print2(){ writeln(NAME2); }
}
```

- `STRUCT_NAME` is the name of this struct. This must be unique within the script.
- pub should be prefixed to members to make them publically accessible
- `DATA_TYPE1` is the data type for the first value.
- `NAME1` is the name for the first value

Keep in mind that recursive dependency is not possible in structs. However you can have an array of the same type inside the struct, as arrays are initialised to length 0.

`this` constructor

Similar to scripts, structs have a constructor that is called after initialising variables:

```
struct Position{
    var int x = 0, y = 0;
    pub this(){
        writeln("constructed 0,0");
    }
    pub this(int x, int y){
        this.x = x;
        this.y = y;
        writeln("constructed with values");
    }
}
```

Enums

Enum can be used to group together constants of the same base data type.

Enums are defined like:

```
enum int EnumName{
    member0 = 1,
    member1 = 3,
    member2 = 3, # same value multiple times is allowed
}
```

- `int` is the base data type
- `EnumName` is the name for this enum

Example:

```
public enum int ErrorType{
    FileNotFound = 1, # default value is first one
    InvalidPath = 1 << 1, # constant expression is allowed
    PermissionDenied = 4
}
```

An enum's member's value can be read as: `EnumName.MemberName`, using the member selector operator.

Enums act as data types:

```
var ErrorType err; # initialised to FileNotFound
// or
var auto err = ErrorType.FileNotFound;
```

Variables

Variable Declaration

Variables can be declared like:

```
var [ref] TYPE var0, var1, var2;
```

- TYPE is the data type of the variables, it can be a `char`, `int`, `float`, `bool`, or an array of those types: `int[]`, or `int[][]`...
- `var0`, `var1`, `var2` are the names of the variables. There can be more/less than 3, and are to be separated by a comma.

Value assignment can also be done in the variable declaration statement like:

```
var int var0 = 12, var1 = 24;
```

Variable Assignment

Variables can be assigned a value like:

```
VAR = VALUE;
```

the data type of `VAR` and `VALUE` must match, or be implicitle castable.

In case the `VAR` is an array, then it's individual elements can be modified:

```
VAR[ INDEX ] = VALUE;
```

- `VAR` is the name of the var/array
- `INDEX` , in case `VAR` is an array, is the index of the element to assign a value to
- `VALUE` is the value to assign

And in a case like this, `VAR[INDEX]` must have the same data type as `VALUE`.

In case you want to modify the whole array, it can be done like:

```
var char someChar = 'a';
var char[] someString;
someString = [someChar, 'b', 'c'] ; # you could've also done [someChar] ~ "bc"
```

Variable Scope

Variables are only available inside the "scope" they are declared in. In the code below:

```
var int someGlobalVar;
public fn main(int count){
    var int i = 0;
    while (i < count){
        var int j;
        # some other code
        i = i + 1;
    }
}
```

Varible `i` and `count` are accessible throughout the function. Variable `j` is accessible only inside the `while` block. Variable `someGlobalVar` is declared outside any function, so it is available to all functions defined *inside* the script, as it is `private`.

Shadowing

In case an identifier matches with a library, and with one defined in script, the one declared in the script will be preferred.

If Statements

If statements are written like:

```
if (CONDITION){
    # some code
}else{
    # some (other?) code
}
```

The `else` part is not required. If `CONDITION` is `false`, then, if the else exists, it's executed, if `CONDITION` is `true`, then `# some code` is executed.

It is not necessary that the `# some code` or the `# some (other?) code` be in a block. In case only one statement is to be executed, it can be written like:

```
if (CONDITION)
    # some code, single statement
else
    # some other code, single statement
```

Indentation is not necessary, it can be written like:

```
if (CONDITION)
# some code, single statement
else
# some other code, single statement
```

Nested If Statements

If statements can also be nested inside other if statements, like:

```
if (CONDITION)
    if (OTHER_CONDITION)
        writeln("OTHER_CONDITION and CONDITION were 1");
    else
        writeln("OTHER_CONDITION was 0, CONDITION was 1");
else
    writeln("CONDITION was 0");
```

Loops

While:

While loops are written like:

```
while (CONDITION){
    # some code in a loop
}
```

As long as `CONDITION` is `true`, `# some code in a loop` is executed. And just like if statements, while loops can also be nested

Do While:

Do while loops are writen like:

```
do{
    # some code in a loop
}while (CONDITION);
```

First the code is executed, then if the condition is `true`, it's executed again.

Foreach:

Foreach loops use `iterators` to iterate over members of a value:

```
foreach (ITERATOR_VALUE, VALUE; CONTAINER){
    # do stuff
    writeln("value at " ~ ITERATOR_VALUE.toString() ~ " is " ~ VALUE.toString());
}
```

Arrays already have iterators and can be used:

```
var int[] bunchOfNumbers = getABunchOfNumbers();
foreach (i, number; bunchOfNumbers)
    writeln(i.toString() ~ "th number is " ~ number.toString());
```

Iterator:

A container struct can implement iterator by offering the following functionality:

```
struct Container{
    struct Position{
        pub fn opIncPre(){ ... }
    }
    pub fn ref int opIndex([ref] Position pos) { ... }
    pub fn Position iteratorStart(){ ... }
    pub fn bool iteratorValid([ref] Position pos){ ... }
}
```

For a container to be iteratable, the following needs to be valid:

- `container.iteratorStart()` should return `something`
- `container.iteratorValid(something)` should return true if something is valid
- `container[something]` should return element, or a ref to it to allow writing
- `++something` should move the `something` to next element

For example, a simple array iteration would be implemented like:

```
struct ArrayInt{
    pub var int[] array;
    pub fn ref int opIndex(int index){
        return array[index];
    }
    pub fn int iteratorStart(){ return 0; }
    pub fn int iteratorValid(int index){
        return index >= 0 && index < array.length;
    }
}
```

Break & Continue

A `break;` statement can be used to exit a loop at any point, and a `continue;` will jump to the end of current iteration.

Operators

Syntax for binary operators is: `A operator B`. The whitespace between operator and A/B is not necessary.

Syntax for unary operators is: `operator A`. Whitespace between operator and A is not necessary.

Operators are read in this order (higher = evaluated first):

1. `.`, `[`, `(`
2. `a++`, `a--`
3. `!a`, `++a`, `--a`
4. `*`, `/`, `%`
5. `+`, `-`, `~`
6. `<<`, `>>`
7. `==`, `!=`, `>=`, `<=`, `>`, `<`, `is`, `!is`
8. `&`, `|`, `^`
9. `&&`, `||`
10. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `~=`, `&=`, `|=`, `^=`

Operator Overloading

Operators are read as functions, and can be overrided same as functions.

Binary operators are translated as:

```
a + b;
// translated to
a.opAdd(b);
```

And unary operators are translated as:

```
a ++;
// translated to
a.opIncPost();
```

The operands for such functions should ideally be `ref`, however it is not necessary.

As such, opAdd and `opIncPost` for a (assuming it to be of type struct A) can be overloaded in following ways:

Member Function

```
struct A{
    pub fn opAdd(ref A right){ ... }
    pub fn opIncPost(){ ... }
}
```

Global Scope Function

```
fn opAdd(ref A left, ref A right) { ... }
fn opIncPost(ref A operand) { ... }
```

Operator Functions

The function associated with each operator is as follows:

(Ta, Tb, T refers to data types, of a,b, or return value)

Operators	Function
.	T opMemberSelect(Ta a, char[] name)
a[b]	T opIndex(Ta a, Tb b)
a(..)	T opFnCall(Ta a, ..)
a++	T opIncPost(Ta a)
a--	T opDecPost(Ta a)
!	T opBoolNot(Ta a)
++a	T opIncPre(Ta a)
--a	T opDecPre(Ta a)
*	T opMultiply(Ta a, Tb b)
/	T opDivide(Ta a, Tb b)
%	T opMod(Ta a, Tb b)
+	T opAdd(Ta a, Tb b)
-	T opSubtract(Ta a, Tb b)
~	T opConcat(Ta a, Tb b)
<<	T opBitshiftLeft(Ta a, Tb b)
>>	T opBitshiftRight(Ta a, Tb b)
==	int opCmp(Ta a, Tb b)
!=	int opCmp(Ta a, Tb b)
>=	int opCmp(Ta a, Tb b)
<=	int opCmp(Ta a, Tb b)
>	int opCmp(Ta a, Tb b)
<	int opCmp(Ta a, Tb b)
&	T opBinAnd(Ta a, Tb b)
|	T opBinOr(Ta a, Tb b)
^	T opBinXor(Ta a, Tb b)
&&	bool opBoolAnd(Ta a, Tb b)
||	bool opBoolOr(Ta a, Tb b)
=	T opAssign(Ta a, Tb b)
+=	T opAddAssign(Ta a, Tb b)
-=	T opSubtractAssign(Ta a, Tb b)
*=	T opMultiplyAssign(Ta a, Tb b)
/=	T opDivideAssign(Ta a, Tb b)
%=	T opModAssign(Ta a, Tb b)
~=	T opConcatAssign(Ta a, Tb b)
&=	T opBinAndAssign(Ta a, Tb b)
|=	T opBinOrAssign(Ta a, Tb b)
^=	T opBinXorAssign(Ta a, Tb b)

the `is`, and `!is` operators are a language feature and do not translate to a function call.

`int opCmp(Ta a, Tb b)`

This function is used to evaluate the >=, <=, >, and < operators.

It should return only one of these values:

- `-1` - a is less than b
- `0` - a is equal to b
- `1` - a is greater than b

The comparison operators are translated as follows:

Expression	**Compiled Form**
`a >= b`	`opCmp(a, b) !is -1`
`a <= b`	`opCmp(a, b) !is 1`
`a > b`	`opCmp(a, b) is 1`
`a < b`	`opCmp(a, b) is -1`