C$ Language Documentation

C$ is intended to be a bit of a cross between the features of C# and the semantics of C++. Currently, C$ is still in the very early stages, and so only supports a subset of the functionalities of C.

Program Structure

A C$ program, much like in most languages, starts with a main function:

```
/* Block comments! */

// My first C$ program
func main() int32 {
    printf("Hello World!");

    return 0;
}
```

As we can see, code commends function similarly to the ones in C++ and C#.

Writing to stdout

C$ has a `printf` function that works more or less identically to the one found in C.

Variable Declarations

We wished for C$ to be very explicit in the mutability of objects, so every variable declaration begins with a mutability specifier, `mut` or `const`. In addition, we have opted to use `:=` as the assignment operator. Putting those two together, a variable declaration will generally look like this:

```
mut int32 a := 5;
```

The language currently supports only a few primitives types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `float32`, `float64`, `bool`, and `string`.

Function Declarations

The main particularities of C$ function definitions are that they begin with the `func` keyword and end with the return type of the function, if there is one. Function parameters are required to have a mutability specifier, even if they are passed by value. While this may seem strange, it opens the door for some optimizations, such as only doing a shallow copy of an object if it is passed as a `const` value.

```
func max(const int32 nb1, const int32 nb2) int32 {
    if nb1 > nb2 {
        return nb1;
    }
```

```
    return nb2;
}

func max(const float64 nb1, const float64 nb2) float64 {
    if nb1 > nb2 {
        return nb1;
    }
    return nb2;
}
```

As one may notice, function overloading is supported by the language.

Note: the declaration order of functions does not matter.

If Statements

The only difference with the `if` in C, C++, or C# is that parentheses are not required for the condition expression. This is true of the loop statements as well.

```
if a < 5 {
    // ...
}
else if a < 10 {
    // ...
}
else {
    // ...
}
```

While Loops

```
while true {
    /* body */
}
```

For Loops

```
for mut int32 i := 0; i < 10; i += 1 {
    /* body */
}
```

Operators

These are the base operators currently supported by the language, listed with decreasing priority:

Operator	Name
`-` `!`	Unary minus and not
`*` `/` `%`	Multiply, Divide, Modulo
`+` `-`	Addition, Subtraction

`<` `>` `<=` `>=`	Comparison
`==` `!=`	Equality
`&`	Bitwise And
`^`	Bitwise Xor
`|`	Bitwise Or
`&&`	And
`||`	Or
`:=`	Assignment
`*=` `/=` `%=` `+=` `-=` `&=` `^=` `|=`	Compound assignment

Of note is that the unary `!` operator pulls double duty as the bitwise Not operator when used on integer types instead of the more traditional `~`.

Structs and Classes

Both structs and classes are declared in a similar fashion as in C++:

```
struct MyStruct {
    bool some_field;
    int32 some_other_field;
}
```

The language (currently) has no constructors or destructors, so structs and classes may be instantiated with a struct literal such as

```
MyStruct{some_field: true, some_other_field: 5}
```

There are no visibility specifiers like `public` and `private`, so everything is always `public`.

Methods

Methods are declared within a struct or class much like in C++ or C#. They are very similar to functions, with the important distinction that they are marked with a mutability qualifier like so:

```
struct Foo {
    Bar field_bar;

    func const do_thing() {
        // ...
    }
}
```

The possible qualifiers are `const` and `mut`, and they denote whether the hidden `this` parameter of the function is a `const ref Foo` or a `mut ref Foo`. In addition, you can use `static` in place of a mutability qualifier.

Arrays

Arrays in C$ are a lie. They are conceptually the same as

```
struct Array<T> {
    uint32 length;
    ptr<T> __ptr__;
}
```

What happens when you create an array literal like

```
const []int32 array := [1, 2, 3];
```

is that an array is allocated on the stack, and then a `struct` is allocated whose `__ptr__` field is set to the address of the actual array. This means that passing an array by value is really just copying a pointer, so it is effectively being passed by reference. Consequently, passing a `ref []T` to a function is pointless.

It is also a bad idea to return an array for this reason, as it would have been allocated on the stack of the function you are returning from.

Strings

C$ doesn't truly have a string type. String literals like `"hello, world!"` are really just a primitive array of characters, or `const []uint8`. One day though!