Object Oriented TEX: TEX♯

TEX♯ *v0.0.2*

1. An Introduction

TEX♯ provides macros which implement object oriented programming in TEX. Suppose you have the following C++ code:

```
struct Person {
    std::string name;

    Person(std::string name) {
        this->name = name;
    }

    void print() {
        printf("My name is %s", this->name);
    }
};
```

(Any suboptimal code design above is intentional (;p) to demonstrate the similarities between C++ and TEX♯.) This can be translated into TEX♯:

```
\class\Person

\method\Person::\Person{\self,\n}{%
    \String\self.\name{\n}%
}

\method\Person::\print{\self}{%
    My name is \self.\name.\print{}%
}
```

A few key things to note is that: a) the definition of a class occurs "outside" of the class body (quotation marks because there is no "class body" in TEX♯), b) methods accept *named* parameters as opposed to the standard numbered parameters required by TEX and c) while TEX♯ borrow's C++'s syntax for declaring member functions of a class, it uses python's method of explicitly passing a reference to the object to its methods.

But overall the syntax shouldn't be too exotic for those who are familiar with C++ and python. Let's now go over the features that TEX♯provides.

2. TEX♯

2.1. Functions

While TEX♯ implements classes and objects, a vital piece of this are functions. An TEX♯ function is very similar to a TEX macro and is in fact a simple wrapper around it. Functions allow you to name parameters and take an arbitrary number of them. For example:

```
\function\foo{\a,\b}{%
    The first parameter is \a, and the second is \b%
}

\foo{Hello,there}
```

Will output `The first parameter is Hello, and the second is there`.

Functions are not scoped by default (since TEX's groups provide very minimal scoping, I decided to forgo it entirely) and therefore any change inside the function affect the program as a whole. But this is not the case for parameters passed to the function, for example:

```
\def\a{The a macro}
\foo{First,Second}

\a
```

Still outputs `The a macro`.

The way this is implemented is quite simple. When you define a function, it creates an array of of the macros you passed it (so in the example above, this array would be `{\a, \b}`), let's call this array the `parameter array`. It then creates a new macro whose name is the name you passed (`\foo`) and prepends code which creates an array of the parameters you passed (`{Hello, there}`) and then redefines the macros in the `parameter array` to be the material you passed the function when calling it. Before it redefines each macro, it saves them in a temporary register which it then uses to redefine them to their previous state after the code has been executed. Due to this, functions should not be recursive, as this would override the register and the macros would not get their original definitions after execution. Hopefully this will change in the future.

The arrays used here are what I called "primitive arrays", or `p@arrays`. Their implementation is similar to Knuth's implementation in Appendix D of his TEXbook. All the code for this can be found in `functions.tex`.

2.2. Scoping

While TEX provides a primitive notion of scoping through grouping, many times this is not sufficient. The main issue with grouping is that either every change is global or local, there is no in between. Suppose you'd like to TEX something similar to:

```
string foo = "foo";
{
    string foo = "bar";
    {
        foo = "bam";
        print(foo);
    }
    print(foo);
}
print(foo);
```

This should print `''bam bam foo''`. But if we were to do something similar in TEX:

```
\def\foo{foo}%
{%
    \def\foo{bar}%
    {%
        \def\foo{bam}%
        \foo\
    }%
    \foo\
}%
\foo
```

This will print `''bam bar foo''`. Adding `\global` before the innermost `\def` will output `''bam bam bam''`.

Therefore TEX#provides two macros: `\beginscope` and `\endscope` which provide scoping for macro definitions. They are similar to TEX's grouping macros and characters, but definitions inside the scope are local to the current group, unless specified that they should be local to the current scope. You can specify this with the `\localize` macro before a macro definition (`\def, \let,` etc). So we can convert the above code to:

```
\def\foo{foo}%
\beginscope%
    \localize\def\foo{bar}%
    \beginscope%
        \def\foo{bam}%
        \foo\
    \endscope%
    \foo\
\endscope
\foo
```

And this will output `''bam bam foo''` just like it should.

Scoping is implemented through a stack. When you `\localize` a macro, its name and definition are pushed onto the stack, and a counter for the number of elements in the current scope is incremented. When you begin a scope, this counter is pushed onto the stack and set to 1, and when you end a scope if the counter is N, then N elements are popped from the stack, and the macros are redefined according to their previous definition.

Now, while this may seem simple, the question arises of how to implement such a stack. The real issue is how you push the definition of the macro onto the stack. Therefore the stack is implemented more as a map than a stack.

TEX#stores two counters relating to the stack: the *global stack counter* and the `local stack counter`. The local stack counter is the counter discussed above, it counts the number of macros on the stack in the current scope. The global stack counter is the number of macros on the stack as a whole. Suppose the global counter is G and the local counter is L.

When a macro `\foo` is `\localized`, two new macros are created: `\@scope@stack@element@`G which is set to `foo`, and `\@scope@stack@def@`G which is let to be equal to `\foo`. Then G and L are incremented. When you call `\endscope`, L elements are popped off the stack, that means that we let the macro whose name is `\@scope@stack@element@`N to be `\@scope@stack@def@`N for N between $G-1$ and $G-L$. Note that since L is pushed onto the stack at the beginning of a scope, this reverts it to its previous definition as well.

2.3. Classes and Objects

Now we get to the meat of this project: the actual object orientation. As you probably know (and if you don't you should read up on object oriented programming before reading this manual as it will not cover it) objects are instances of what are called *classes*. So naturally TEX#must implement a method of creating these classes, which it does through the macro `class`.

`\class ⟨class name⟩`

This creates a class whose name is `⟨class name⟩`, this must be macro token. In order to define member functions/methods for a specific class, you must use the `method` macro:

`\method ⟨class name⟩::⟨method name⟩{⟨parameters...⟩}{⟨code⟩}`

This will create a method for the class `class name` whose name is `method name` which accepts `parameters...` as parameters and executes `code` upon calling. `parameters...` must be comma-delimited, and note that it is space-insensitive and the first parameter must refer to a reference of the current instance of the class. So for example the following is a valid method creation:

```
\method\Person::\speak{\self,\speech}{%
    \self.\name >> \speech%
}
```

Member variables are not explicitly created in the class body and are instead created inside a method of the object. So for examlple if we'd like to create a `Person` instance inside some object's method called `\person` whose name is "Fin" then we can do `\Person\self.\person{Fin}` inside an object method.

You can also define a constructor for classes by creating a method whose name is the same as the class's. For example we can define:

```
\class\Foo

\method\Foo::\Foo{\self,\f}{%
    Constructing a Foo instance with parameter \f%
}

\Foo\foo{bar}
```

This will output `Constructing a Foo instance with parameter bar`.

The `\class` macro creates a new macro by the name of the instance name passed to it, as well as a construction macro for creating new instances of this class. For purposes of this explanation, let's suppose this name is `\Foo`. So `\class\Foo` will create a macro called `\Foo` whose sole job is to initiate the construction process for a new `\Foo` instance. The construction process first gets the name of the instance to create, which may be slightly complicated if you have members of members and references. For example, suppose you're in an instance `\instance`'s method, doing something like:

```
\Foo\self.\member.\foo ...
```

Must convert this into the "justified" name `instance::member::foo`.

As mentioned above, member variables have the form `instance::member`. Similarly, member functions are simply functions whose name is of the form `class::method`. So a call to `\method\Foo::\bar...` simply creates a function whose name is `Foo::bar`. In fact this is equivalent to `\expandafter\function\csname Foo::bar\endcsname ...`.

The next step in the comstruction process is the actual construction of the instance. As said briefly above, every class has a construction macro which creates new instances. Once the program has the justified name of the new instance to be created, it passes the name to the class's construction macro. The construction macro creates the new instance by defining a new macro whose name is this justified name, and then passes a reference of this new instance to the construction method (`\Foo::\Foo`).

The macro which is created for the instance is a simple macro which just checks the tokens that come after it. If the macro is followed by a period, then it replaces the call to the name of the instance's member variable (`\foo.\member` is replaced by `\foo::member`). And if the next token is an open brace (`{`) then it replaces the call to a call to the class's method of that name and passes the instance as the first parameter to the method (`\foo.\bar{}` is replaced by `\Foo::bar{\foo}`)

Every instance also recongizes another token: `?`. This is called the *query suffix*. If you place query suffix after an instance it will place its actual name into a register called `\tpp@instance@name`. So for example if you have an instance `\foo` and you call `\foo.\bar`, the macro `\self` will be a reference to `\foo`, and without the query suffix you would have no way of getting the actual name of the instance. So if you were to call `\self.\member` how would it know what macro to look for? So calling `\self?` will place `foo` into `\tpp@instance@name`, and this can be used for lookups. A more versatile alternative is to use the `\@get@instance@name` macro, which will set `\tpp@instance@name` to the name of the reference if it doesn't point anywhere (if it is undefined).

Classes also support limited operator overloading using the macro `\operator`. Its use is identical to the `\method` macro but instead of getting a method name it gets an operator to overload. (For now this operator can only be a single token.)

So for example:

```
\operator\Foo::+{\self,\other}{%
    Adding two Foo instances%
}

\Foo\fooA{}
\Foo\fooB{}

\fooA + \fooB
```

Will output `Adding two Foo instances`.

An operator overload is also a method, so for example the name of the one above is `\Foo::+`. At first, implementing this may seem like a breeze, or at least similar to implementing `\method`. The issue is when you have code like `\foo+ \self.\foo`, you somehow need to first parse the `\self.\foo` into `\instance::foo` and then pass it to the operator. In order to do this, it utilizes the same process used for construction. As a result of this, the justified name of `\other` is stored in the register `\tpp@instance@name` upon calling the operator.

TEX# also provides a (minimal as of now) standard library with predefined classes, the String and Num class. The String class represents a string and can be constructed, assigned (operator =), concatenated (operator +), and printed (using the `\print` method). For example:

```
\String\strA{Hello}
\String\space{ }
\String\strB{world!}

\strA + \space
\strA + \strB
\strA.\print{}      % Hello world!

\strA = \strB
\strA.\print{}      % world!
```

Num represents an integer and can be constructed, added, multiplied, divided, assigned, and printed. For example:

```
\Num\numA{10}
\Num\numB{7}

\numA * \numB
\numA.\print{}      % 70

\Num\numC{3}
\numB / \numC
\numB.\print{}      % 2 (integer division)
```