```
class Messenger {
    message = "Hello World";
    start() {
        setTimeout(() => alert(this.message), 3000);
    }
};
var messenger = new Messenger();
messenger.start();
```

the use of an arrow function expression causes the callback to have the same `this` as the surrounding 'start' method. Writing the callback as a standard function expression it becomes necessary to manually arrange access to the surrounding `this`, for example by copying it into a local variable:

```
class Messenger {
    message = "Hello World";
    start() {
        var _this = this;
        setTimeout(function() { alert(_this.message); }, 3000);
    }
};
var messenger = new Messenger();
messenger.start();
```

The TypeScript compiler applies this type of transformation to rewrite arrow function expressions into standard function expressions.

A construct of the form

```
< Identifier > ( ParamList ) => { ... }
```

could be parsed as an arrow function expression with a type parameter or a type assertion applied to an arrow function with no type parameter. It is resolved as the former, but parentheses can be used to select the latter meaning:

```
< Identifier > ( ( ParamList ) => { ... } )
```

4.9.3 Contextually Typed Function Expressions

Function expressions with no type parameters and no parameter or return type annotations (but possibly with optional parameters and default parameter values) are contextually typed in certain circumstances, as described in section 4.19.

When a function expression is contextually typed by a function type *T*, the function expression is processed as if it had explicitly specified parameter type annotations as they exist in *T*. Parameters are matched by position and need not have matching names. If the function expression has fewer parameters than *T*, the additional parameters in *T* are ignored. If the function expression has more parameters than *T*, the additional parameters are all considered to have type Any.