

# The Type Description Language (TDL)—Design Document

Eric Caraszi and Luke Hodorowicz  
<http://www.cs.rpi.edu/research/gpg/Simplicissimus>

## 1 Purpose

The Type Description Language was created in order to simplify the conveyance of information about library functions to the Simplicissimus software. The compiler needs to know the form of functions (return value, arguments, constant modifiers) and certain semantic information (exceptions, side effects).

This document will describe the design goals of the Type Description Language, the structure of a TDL file, and the semantics of statements therein, the grammar, and an example.

## 2 Design Goals

- **Ease of use** Simple text script files, simple command line processing. Ability to be used with standard build tools (make).
- **Readability** Script should be generally understandable even to someone not familiar with the simplifier.
- **Principle of Least Typing** No information should be entered twice.
- **Extensible** Implementation should be simple and understandable. Addition of flags or other information should be straightforward.

## 3 Description

### 3.1 Structure

A TDL file contains a series of traits definitions. A trait definition is defined as:

```
<function prototype> [ <flags list> ] <operator class> ;
```

Trait definitions of class member functions are enclosed in a class statement:

```
class <class name> {  
    <function prototype> [ <flags list> ] <operator class> ;  
    ...  
};
```

Function prototypes are defined as in C++, using parameter identifiers as described below. See also **Constructors**, below.

The flags list is empty, or contains flag(s) separated by whitespace.

Traits definitions must have an operator class name, to link it semantically into the simplifier.

The keyword `const` may appear at the end of a function prototype, indicating in member functions that the class instance is not modified.

## 3.2 Parameter Identifiers

Parameters, return values, class names, function names, and operator class names all are specified by parameter identifiers.

A parameter identifier is any normal C++ identifier, with template arguments or scope resolution as needed. Reference or pointer (`&`, `*`) qualifiers are allowed.

If a parameter identifier includes any spaces (such as needed with complex template arguments) the entire identifier must be enclosed by `$. . .$`. This is meant to avoid having to parse complex C++ type names. For example, `std::complex<T>&` and `$std::complex< T > &$` are the same identifier. Future implementations may include bracket counting to handle simple cases.

The keyword `const` may appear before any parameter identifier to indicate the corresponding [argument, return value] is constant.

## 3.3 DEFINE semantics

```
DEFINE <identifier> [ <argument-list> ]
```

`DEFINE` creates a set of arguments to be used in place of a parameter identifier in a trait definition. Multiple “instances” of the trait definition (i.e. overloaded functions) will be expanded across the set of arguments.

Multiple `DEFINE` statements with the same identifier produces undefined behavior.

Use the identifier in place of each argument to be expanded into. The same `DEFINE` can be used multiple times in the same trait definition. The scope of a `DEFINE` begins at its definition and continues through to the end of the file.

Multiple `DEFINES` used in the same function must contain the same number of arguments. The sets of arguments in the trait definition will define the different instances. For example, the sequence:

```
DEFINE s [ a, b, c ]
DEFINE t [ d, e, f ]
int add(s, t) [FLAGS] add_expr;
```

will effectively expand to:

```
int add(a, d) [FLAGS] add_expr;
int add(b, e) [FLAGS] add_expr;
int add(c, f) [FLAGS] add_expr;
```

DEFINE is implemented in the script processor and should not be considered a macro or preprocessor step.

### 3.4 TEMPLATEPARAMS semantics

```
TEMPLATEPARAMS [ <param list> ]
```

A TEMPLATEPARAMS statement describes what template parameters are included in the parameter identifiers after it. The scope of the TEMPLATEPARAMS defined is from the point of definition until the next TEMPLATEPARAMS statement, or the end of the file.

Template parameters are lists of valid parameter identifiers, separated by commas. Single template parameters (“T” “X”) are assumed to be C++ *typenamees*. Template parameters with at least two terms (“int X”, “double Y”) are *nontype* parameters and are passed literally.

### 3.5 Miscellany

**Constructors** Constructors are different only in that they have no return value (as in C++). Constructor trait definitions must be included in their class block.

## 4 Example

This is a comment:

```
#  
# TDL file for std::complex class  
#
```

The following specifies a few simple global functions and operators. Note the TEMPLATEPARAMS defining T as a template parameter.

```
TEMPLATEPARAMS [ T ]
```

```
std::complex<T> operator + (const std::complex<T>&) [PREFIX APPLICATIVE] UnaryPlus;  
std::complex<T> operator - (const std::complex<T>&) [PREFIX APPLICATIVE] UnaryMinus;
```

```
std::complex<T> conj(const std::complex<T>&) [APPLICATIVE FUNC] Conj;  
std::complex<T> cos(const std::complex<T>&) [APPLICATIVE FUNC] Cos;  
std::complex<T> cosh(const std::complex<T>&) [APPLICATIVE FUNC] Cosh;  
std::complex<T> exp(const std::complex<T>&) [APPLICATIVE FUNC] Exp;  
std::complex<T> log(const std::complex<T>&) [APPLICATIVE FUNC] Log;  
std::complex<T> log10(const std::complex<T>&) [APPLICATIVE FUNC] Log10;  
std::complex<T> sin(const std::complex<T>&) [APPLICATIVE FUNC] Sin;  
std::complex<T> sinh(const std::complex<T>&) [APPLICATIVE FUNC] Sinh;  
std::complex<T> sqrt(const std::complex<T>&) [APPLICATIVE FUNC] Sqrt;  
std::complex<T> tan(const std::complex<T>&) [APPLICATIVE FUNC] Tan;  
std::complex<T> tanh(const std::complex<T>&) [APPLICATIVE FUNC] Tanh;
```

```

T real(const std::complex<T>&) [APPLICATIVE FUNC] Real;
T imag(const std::complex<T>&) [APPLICATIVE FUNC] Imag;
T abs(const std::complex<T>&) [APPLICATIVE FUNC] Abs;
T arg(const std::complex<T>&) [APPLICATIVE FUNC] Arg;
T norm(const std::complex<T>&) [APPLICATIVE FUNC] Norm;

std::complex<T> polar(const T&, const T&) [APPLICATIVE FUNC] Polar;
std::complex<T> pow(const std::complex<T>&, int) [APPLICATIVE FUNC] Pow;

```

Here DEFINE is used to save re-typing a set of operators overloaded over three pairs of arguments.

```

DEFINE cpxcpxT [ const std::complex<T>&, const std::complex<T>&, const T& ]
DEFINE Tcpxcpx [ const T&, const std::complex<T>&, const std::complex<T>& ]

std::complex<T> operator * (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE OVERFLOW UNDERFLOW] Mul;
std::complex<T> operator / (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE ZERO_DIVIDE UNDERFLOW] Div;
std::complex<T> operator + (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE OVERFLOW] Add;
std::complex<T> operator - (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE OVERFLOW] Sub;
std::complex<T> pow      (cpxcpxT, Tcpxcpx) [APPLICATIVE OVERFLOW UNDERFLOW FUNC] Pow;

bool operator == (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE] Equal;
bool operator != (cpxcpxT, Tcpxcpx) [INFIX APPLICATIVE] NotEqual;

```

Here the member function traits are specified. TEMPLATEPARAMS specifies both possible template arguments when needed, and DEFINE expands the overloaded operators. Also note the constructor specification, where `...$` is needed to delimit the `complex` template argument in the operator class.

```

TEMPLATEPARAMS [ T, X ]

DEFINE TcpxTcpxX [ const T&, const std::complex<T>&, const std::complex<X>& ]

class std::complex<T> {

    std::complex<T> operator = (TcpxTcpxX) [INFIX SIDE_EFFECTS] Assn;
    std::complex<T> operator *= (TcpxTcpxX) [INFIX SIDE_EFFECTS OVERFLOW UNDERFLOW] MulAssn;
    std::complex<T> operator /= (TcpxTcpxX) [INFIX SIDE_EFFECTS ZERO_DIVIDE UNDERFLOW] DivAssn;
    std::complex<T> operator += (TcpxTcpxX) [INFIX SIDE_EFFECTS OVERFLOW] AddAssn;
    std::complex<T> operator -= (TcpxTcpxX) [INFIX SIDE_EFFECTS OVERFLOW] SubAssn;

    # constructor
    complex(const std::complex<X>&) [FUNC] $Construct<std::complex<T>> >$;

TEMPLATEPARAMS [ T ]

    # more constructors
    complex(const std::complex<T>&) [FUNC] $Construct<std::complex<T>> >$;
    complex(const T&, const T&) [FUNC] $Construct<std::complex<T>> >$;

};

```

## 5 Grammar

*stmt-list:*

```
stmt-list traits-def
traits-def
stmt-list class-def
class-def
stmt-list define-stmt
define-stmt
stmt-list template-param-stmt
template-param-stmt
```

*traits-def:*

```
func-spec [ flags-list ] param-ident ;
```

*class-def:*

```
class param-ident { stmt-list } ;
```

*define-stmt:*

```
DEFINE identifier [ param-ident-list ]
```

*template-param-stmt:*

```
TEMPLATEPARAMS [ param-ident-list-list ]
```

*func-spec:*

```
func-proto
func-proto const
```

*func-proto:*

```
param-spec param-ident ( arg-list )
param-spec operator op-id ( arg-list )
param-ident ( arg-list )
```

*arg-list:*

```
arg-list , param-spec
param-spec
λ
```

*param-spec:*

```
param-ident-wconst
identifier
```

*param-ident-wconst:*

```
param-ident
const param-ident
```

*param-ident-list:*

```
param-ident-list , param-ident-wconst
param-ident-wconst
```

*param-ident-list-list:*

```
param-ident-list , param-ident-wconst
param-ident-list param-ident-wconst
param-ident-wconst
```

*param-ident:*

\$.+\$  
[a-zA-Z\_]+[a-zA-Z0-9<>\_]\*

*identifier:*  
[a-zA-Z\_]+[a-zA-Z0-9\_]\*

*op-id:*  
+ - \* / % << >> & | ^ ~ ,  
+= -= \*= /= %= <<= >>= &= |= ^= ~=  
&& || > >= < <= == != !  
++ -- , ->\* -> () []  
new delete new[] delete[]

*flags-list:*  
*flags-list* *flag*  
*flag*  
 $\lambda$

*flag:*  
APPLICATIVE  
SIDE\_EFFECTS  
ZERO\_DIVIDE  
OVERFLOW  
UNDERFLOW  
PREFIX  
FUNC