

# A C++ interface to SWI-Prolog

Jan Wielemaker  
SWI,  
University of Amsterdam  
The Netherlands  
E-mail: `jan@swi.psy.uva.nl`

July 3, 2009

## Abstract

This document describes a C++ interface to SWI-Prolog. SWI-Prolog could be used with C++ for a very long time, but only by calling the extern "C" functions of the C-interface. The interface described herein provides a true C++ layer around the C-interface for much more concise and natural programming from C++. The interface deals with automatic type-conversion to and from native C data-types, transparent mapping of exceptions, making queries to Prolog and registering foreign predicates.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Overview</b>	<b>4</b>
<b>3</b>	<b>Examples</b>	<b>5</b>
3.1	Hello(World) . . . . .	5
3.2	Adding numbers . . . . .	6
3.3	Average of solutions . . . . .	6
<b>4</b>	<b>The class PTerm</b>	<b>7</b>
4.1	Constructors . . . . .	7
4.2	Casting PTerm to native C-types . . . . .	8
4.3	Unification . . . . .	8
4.4	Comparison . . . . .	9
4.5	Analysing compound terms . . . . .	9
4.6	Miscellaneous . . . . .	10
4.7	The class PString . . . . .	10
4.8	The class PCodeList . . . . .	10
4.9	The class PCharList . . . . .	11
4.10	The class PCompound . . . . .	11
4.11	The class PTail . . . . .	11
<b>5</b>	<b>The class PTermv</b>	<b>12</b>
<b>6</b>	<b>Supporting Prolog constants</b>	<b>13</b>
<b>7</b>	<b>The class PRegister</b>	<b>14</b>
<b>8</b>	<b>The class PQuery</b>	<b>15</b>
8.1	The class PFrame . . . . .	16
<b>9</b>	<b>The PREDICATE macro</b>	<b>17</b>
9.1	Controlling the Prolog destination module . . . . .	17
<b>10</b>	<b>Exceptions</b>	<b>18</b>
10.1	The class PException . . . . .	18
10.2	The class PTypeError . . . . .	19
10.3	The class PIDomainError . . . . .	19
<b>11</b>	<b>Embedded applications</b>	<b>20</b>
<b>12</b>	<b>Considerations</b>	<b>20</b>
12.1	The C++ versus the C interface . . . . .	20
12.2	Static linking and embedding . . . . .	20
12.3	Status and compiler versions . . . . .	21
12.4	Limitations . . . . .	21



## 1 Introduction

C++ provides a number of features that make it possible to define a much more natural and concise interface to dynamically typed languages than plain C does. Using programmable type-conversion (*casting*), native data-types can be translated automatically into appropriate Prolog types, automatic destructors can be used to deal with most of the cleanup required and C++ exception handling can be used to map Prolog exceptions and interface conversion errors to C++ exceptions, which are automatically mapped to Prolog exceptions as control is turned back to Prolog.

### Competing interfaces

Volker Wyk has defined an alternative C++ mapping based on templates and compatible to the STL framework. See <http://www.volker-wyk.de/swiprolog-c++/index.html>.

### Acknowledgements

I would like to thank Anjo Anjewierden for comments on the definition, implementation and documentation of this package.

## 2 Overview

The most useful area for exploiting C++ features is type-conversion. Prolog variables are dynamically typed and all information is passed around using the C-interface type `term_t`. In C++, `term_t` is embedded in the *lightweight* class `PlTerm`. Constructors and operator definitions provide flexible operations and integration with important C-types (`char *`, `long` and `double`).

The list below summarises the classes defined in the C++ interface.

#### Class `PlTerm`

Generic Prolog term. Provides constructors and operators for conversion to native C-data and type-checking.

#### Class `PlString`

Subclass of `PlTerm` with constructors for building Prolog string objects.

#### Class `PlCodeList`

Subclass of `PlTerm` with constructors for building Prolog lists of ASCII values.

#### Class `PlCharList`

Subclass of `PlTerm` with constructors for building Prolog lists of one-character atoms (as `atom_chars/2`).

#### Class `PlCompound`

Subclass of `PlTerm` with constructors for building compound terms.

#### Class `PlTail`

SubClass of `PlTerm` for building and analysing Prolog lists.

**Class PLTermv**

Vector of Prolog terms. See `PL_new_term_refs()`. the `[]` operator is overloaded to access elements in this vector. `PLTermv` is used to build complex terms and provide argument-lists to Prolog goals.

**Class PLException**

Subclass of `PLTerm` representing a Prolog exception. Provides methods for the Prolog communication and mapping to human-readable text representation.

**Class PLTypeError**

Subclass of `PLException` for representing a Prolog `type_error` exception.

**Class PLDomainError**

Subclass of `PLException` for representing a Prolog `domain_error` exception.

**Class PLAtom**

Allow for manipulating atoms in their internal Prolog representation for fast comparison.

**Class PLQuery**

Represents opening and enumerating the solutions to a Prolog query.

**Class PLFrame**

This utility-class can be used to discard unused term-references as well as to do ‘*data-backtracking*’.

**Class PLEngine**

This class is used in *embedded* applications (applications where the main control is held in C++). It provides creation and destruction of the Prolog environment.

**Class PLRegister**

The encapsulation of `PL_register_foreign()` is defined to be able to use C++ global constructors for registering foreign predicates.

The required C(++) function header and registration of a predicate is arranged through a macro called `PREDICATE()`.

## 3 Examples

Before going into a detailed description of the C++ classes we present a few examples illustrating the ‘feel’ of the interface.

### 3.1 Hello(World)

This very simple example shows the basic definition of the predicate `hello/1` and how a Prolog argument is converted to C-data:

```
PREDICATE(hello, 1)
{ cout << "Hello " << (char *)A1 << endl;

  return TRUE;
}
```

The arguments to `PREDICATE()` are the name and arity of the predicate. The macros `A<n>` provide access to the predicate arguments by position and are of the type `PlTerm`. Casting a `PlTerm` to a `char *` provides the natural type-conversion for most Prolog data-types, using the output of `write/1` otherwise:

```
?- hello(world).
Hello world

Yes
?- hello(X)
Hello _G170

X = _G170
```

### 3.2 Adding numbers

This example shows arithmetic using the C++ interface, including unification, type-checking and conversion. The predicate `add/3` adds the two first arguments and unifies the last with the result.

```
PREDICATE(add, 3)
{ return A3 = (long)A1 + (long)A2;
}
```

Casting a `PlTerm` to a `long` performs a `PL_get_long()` and throws a C++ exception if the Prolog argument is not a Prolog integer or float that can be converted without loss to a `long`. The `=` operator of `PlTerm` is defined to perform unification and returns `TRUE` or `FALSE` depending on the result.

```
?- add(1, 2, X).

X = 3.
?- add(a, 2, X).
[ERROR: Type error: 'integer' expected, found 'a']
Exception: ( 7) add(a, 2, _G197) ?
```

### 3.3 Average of solutions

This example is a bit harder. The predicate `average/3` is defined to take the template `average(+Var, :Goal, -Average)`, where *Goal* binds *Var* and will unify *Average* with average of the (integer) results.

`PlQuery` takes the name of a predicate and the goal-argument vector as arguments. From this information it deduces the arity and locates the predicate. the member-function `next_solution()` yields `TRUE` if there was a solution and `FALSE` otherwise. If the goal yielded a Prolog exception it is mapped into a C++ exception.

```
PREDICATE(average, 3)
{ long sum = 0;
```

```

long n = 0;

PlQuery q("call", PlTermv(A2));
while( q.next_solution() )
{ sum += (long)A1;
  n++;
}
return A3 = (double)sum/(double)n;
}

```

## 4 The class PlTerm

As we have seen from the examples, the `PlTerm` class plays a central role in conversion and operating on Prolog data. This section provides complete documentation of this class.

### 4.1 Constructors

#### **PlTerm::PlTerm()**

Creates a new initialised term (holding a Prolog variable).

#### **PlTerm::PlTerm(*term\_t* t)**

Converts between the C-interface and the C++ interface by turning the term-reference into an instance of `PlTerm`. Note that, being a lightweight class, this is a no-op at the machine-level!

#### **PlTerm::PlTerm(*const char* \*text)**

Creates a term-references holding a Prolog atom representing *text*.

#### **PlTerm::PlTerm(*const PlAtom* &atom)**

Creates a term-references holding a Prolog atom from an atom-handle.

#### **PlTerm::PlTerm(*long* n)**

Creates a term-references holding a Prolog integer representing *n*.

#### **PlTerm::PlTerm(*double* f)**

Creates a term-references holding a Prolog float representing *f*.

#### **PlTerm::PlTerm(*void* \*ptr)**

Creates a term-references holding a Prolog pointer. A pointer is represented in Prolog as a mangled integer. The mangling is designed to make most pointers fit into a *tagged-integer*. Any valid pointer can be represented. This mechanism can be used to represent pointers to C++ objects in Prolog. Please note that ‘myclass’ should define conversion to and from `void *`.

```

PREDICATE (make_my_object, 1)
{ myclass *myobj = new myclass();

  return A1 = (void *)myobj;
}

```

```

PREDICATE (free_my_object, 1)
{ myclass *myobj = (void *)A1;

    delete(myobj);
    return TRUE;
}

```

## 4.2 Casting PTerm to native C-types

PTerm can be casted to the following types:

### **PTerm::operator term\_t(void)**

This cast is used for integration with the C-interface primitives.

### **PTerm::operator long(void)**

Yields a long if the PTerm is a Prolog integer or float that can be converted without loss to a long. throws a `type_error` exception otherwise.

### **PTerm::operator int(void)**

Same as for long, but might represent fewer bits.

### **PTerm::operator double(void)**

Yields the value as a C double if PTerm represents a Prolog integer or float.

### **PTerm::operator char \*(void)**

Converts the Prolog argument using `PL_get_chars()` using the flags `CVT_ALL|CVT_WRITE|BUF_RING`, which implies Prolog atoms and strings are converted to the represented text. All other data is handed to `write/1`. If the text is static in Prolog, a direct pointer to the string is returned. Otherwise the text is saved in a ring of 16 buffers and must be copied to avoid overwriting.

### **PTerm::operator void \*(void)**

Extracts pointer value from a term. The term should have been created by `PTerm::PTerm(void*)`.

## 4.3 Unification

### **int PTerm::operator =(Type)**

The operator = is defined for the *Types* PTerm, long, double, char \* and PAtom. It performs Prolog unification and returns TRUE if successful and FALSE otherwise.

The boolean return-value leads to somewhat unconventional-looking code as normally, assignment returns the value assigned in C. Unification however is fundamentally different to assignment as it can succeed or fail. Here is a common example.

```

PREDICATE (hostname, 1)
{ char buf[32];

    if ( gethostname(buf, sizeof(buf)) == 0 )

```



```

        return A1 = buf;

    return FALSE;
}

```

## 4.4 Comparison

```

int PlTerm::operator ==(const PlTerm &t)
int PlTerm::operator !=(const PlTerm &t)
int PlTerm::operator <(const PlTerm &t)
int PlTerm::operator >(const PlTerm &t)
int PlTerm::operator <=(const PlTerm &t)
int PlTerm::operator >=(const PlTerm &t)

```

Compare the instance with *t* and return the result according to the Prolog defined *standard order of terms*.

```

int PlTerm::operator ==(long num)
int PlTerm::operator !=(long num)
int PlTerm::operator <(long num)
int PlTerm::operator >(long num)
int PlTerm::operator <=(long num)
int PlTerm::operator >=(long num)

```

Convert `PlTerm` to a long and perform standard C-comparison between the two long integers. If `PlTerm` cannot be converted a `type_error` is raised.

```

int PlTerm::operator ==(const char *)

```

Yields `TRUE` if the `PlTerm` is an atom or string representing the same text as the argument, `FALSE` if the conversion was successful, but the strings are not equal and a `type_error` exception if the conversion failed.

Below are some typical examples. See section 6 for direct manipulation of atoms in their internal representation.

<code>A1 &lt; 0</code>	Test <i>A1</i> to hold a Prolog integer or float that can be transformed lossless to an integer less than zero.
<code>A1 &lt; PlTerm(0)</code>	<i>A1</i> is before the term '0' in the 'standard order of terms'. This means that if <i>A1</i> represents an atom, this test yields <code>TRUE</code> .
<code>A1 == PlCompound("a(1)")</code>	Test <i>A1</i> to represent the term <code>a(1)</code> .
<code>A1 == "now"</code>	Test <i>A1</i> to be an atom or string holding the text "now".

## 4.5 Analysing compound terms

Compound terms can be viewed as an array of terms with a name and arity (length). This view is expressed by overloading the `[]` operator.

A `type_error` is raised if the argument is not compound and a `domain_error` if the index is out of range.

In addition, the following functions are defined:

`PLTerm` **PLTerm::operator []**(*int arg*)

If the `PLTerm` is a compound term and *arg* is between 1 and the arity of the term, return a new `PLTerm` representing the *arg*-th argument of the term. If `PLTerm` is not compound, a `type_error` is raised. If *arg* is out of range, a `domain_error` is raised. Please note the counting from 1 which is consistent to Prolog's `arg/3` predicate, but inconsistent to C's normal view on an array. See also class `PLCompound`. The following example tests *x* to represent a term with first-argument an atom or string equal to `gnat`.

```
.../  
if ( x[1] == "gnat" )  
...
```

`const char *` **PLTerm::name**()

Return a `const char *` holding the name of the functor of the compound term. Raises a `type_error` if the argument is not compound.

`int` **PLTerm::arity**()

Returns the arity of the compound term. Raises a `type_error` if the argument is not compound.

## 4.6 Miscellaneous

`int` **PLTerm::type**()

Yields the actual type of the term as `PL_term_type()`. Return values are `PL_VARIABLE`, `PL_FLOAT`, `PL_INTEGER`, `PL_ATOM`, `PL_STRING` or `PL_TERM`

To avoid very confusing combinations of constructors and therefore possible undesirable effects a number of subclasses of `PLTerm` have been defined that provide constructors for creating special Prolog terms. These subclasses are defined below.

## 4.7 The class `PLString`

A SWI-Prolog string represents a byte-string on the global stack. Its lifetime is the same as for compound terms and other data living on the global stack. Strings are not only a compound representation of text that is garbage-collected, but as they can contain 0-bytes, they can be used to contain arbitrary C-data structures.

**PLString::PLString**(*const char \*text*)

Create a SWI-Prolog string object from a 0-terminated C-string. The *text* is copied.

**PLString::PLString**(*const char \*text, int len*)

Create a SWI-Prolog string object from a C-string with specified length. The *text* may contain 0-characters and is copied.

## 4.8 The class `PLCodeList`

**PLCodeList::PLCodeList**(*const char \*text*)

Create a Prolog list of ASCII codes from a 0-terminated C-string.

## 4.9 The class `PlCharList`

Character lists are compliant to Prolog's `atom_chars/2` predicate.

**`PlCharList::PlCharList(const char *text)`**

Create a Prolog list of one-character atoms from a 0-terminated C-string.

## 4.10 The class `PlCompound`

**`PlCompound::PlCompound(const char *text)`**

Create a term by parsing (as `read/1`) the *text*. If the *text* is not valid Prolog syntax, a `syntax_error` exception is raised. Otherwise a new term-reference holding the parsed text is created.

**`PlCompound::PlCompound(const char *functor, PlTermv args)`**

Create a compound term with the given name from the given vector of arguments. See `PlTermv` for details. The example below creates the Prolog term `hello(world)`.

```
PlCompound("hello", PlTermv("world"))
```

## 4.11 The class `PlTail`

The class `PlTail` is both for analysing and constructing lists. It is called `PlTail` as enumeration-steps make the term-reference follow the 'tail' of the list.

**`PlTail::PlTail(PlTerm list)`**

A `PlTail` is created by making a new term-reference pointing to the same object. As `PlTail` is used to enumerate or build a Prolog list, the initial *list* term-reference keeps pointing to the head of the list.

**`int PlTail::append(const PlTerm &element)`**

Appends *element* to the list and make the `PlTail` reference point to the new variable tail. If *A* is a variable, and this function is called on it using the argument `"gnat"`, a list of the form `[gnat|B]` is created and the `PlTail` object now points to the new variable *B*.

This function returns `TRUE` if the unification succeeded and `FALSE` otherwise. No exceptions are generated.

The example below translates the `main()` argument vector to Prolog and calls the prolog predicate `entry/1` with it.

```
int
main(int argc, char **argv)
{ PlEngine e(argv[0]);
  PlTermv av(1);
  PlTail l(av[0]);

  for(int i=0; i<argc; i++)
    l.append(argv[i]);
```

```

l.close();

PlQuery q("entry", av);
return q.next_solution() ? 0 : 1;
}

```

**int PlTail::close()**

Unifies the term with [] and returns the result of the unification.

**int PlTail::next(PlTerm &t)**

Bind *t* to the next element of the list PlTail and advance PlTail. Returns TRUE on success and FALSE if PlTail represents the empty list. If PlTail is neither a list nor the empty list, a `type_error` is thrown. The example below prints the elements of a list.

```

PREDICATE(write_list, 1)
{ PlTail tail(A1);
  PlTerm e;

  while(tail.next(e))
    cout << (char *)e << endl;

  return TRUE;
}

```

## 5 The class PlTermv

The class `PlTermv` represents an array of term-references. This type is used to pass the arguments to a foreignly defined predicate, construct compound terms (see `PlTerm::PlTerm(const char *name, PlTermv arguments)`) and to create queries (see `PlQuery`).

The only useful member function is the overloading of [], providing (0-based) access to the elements. Range checking is performed and raises a `domain_error` exception.

The constructors for this class are below.

**PlTermv::PlTermv(int size)**

Create a new array of term-references, all holding variables.

**PlTermv::PlTermv(int size, term\_t t0)**

Convert a C-interface defined term-array into an instance.

**PlTermv::PlTermv(PlTerm ...)**

Create a vector from 1 to 5 initialising arguments. For example:

```

load_file(const char *file)
{ return PlCall("compile", PlTermv(file));
}

```

If the vector has to contain more than 5 elements, the following construction should be used:

```
{ PlTermv av(10);

  av[0] = "hello";
  ...
```

## 6 Supporting Prolog constants

Both for quick comparison as for quick building of lists of atoms, it is desirable to provide access to Prolog's atom-table, mapping handles to unique string-constants. If the handles of two atoms are different it is guaranteed they represent different text strings.

Suppose we want to test whether a term represents a certain atom, this interface presents a large number of alternatives:

### Direct comparision to char \*

Example:

```
PREDICATE(test, 1)
{ if ( A1 == "read" )
  ...;
```

This writes easily and is the preferred method if performance is not critical and only a few comparisons have to be made. It validates *A1* to be a term-reference representing text (atom, string, integer or float) extracts the represented text and uses strcmp() to match the strings.

### Direct comparision to PlAtom

Example:

```
static PlAtom ATOM_read("read");

PREDICATE(test, 1)
{ if ( A1 == ATOM_read )
  ...;
```

This case raises a `type_error` if *A1* is not an atom. Otherwise it extracts the atom-handle and compares it to the atom-handle of the global `PlAtom` object. This approach is faster and provides more strict type-checking.

### Extraction of the atom and comparison to PlAtom

Example:

```

static PlAtom ATOM_read("read");

PREDICATE(test, 1)
{ PlAtom a1(A1);

    if ( a1 == ATOM_read )
        ...;
}

```

This approach is basically the same as section 6, but in nested if-then-else the extraction of the atom from the term is done only once.

### Extraction of the atom and comparison to char \*

Example:

```

PREDICATE(test, 1)
{ PlAtom a1(A1);

    if ( a1 == "read" )
        ...;
}

```

This approach extracts the atom once and for each test extracts the represented string from the atom and compares it. It avoids the need for global atom constructors.

#### **PlAtom::PlAtom(*atom\_t handle*)**

Create from C-interface atom handle. Used internally and for integration with the C-interface.

#### **PlAtom::PlAtom(*const char \*text*)**

Create from a string. The *text* is copied if a new atom is created.

#### **PlAtom::PlAtom(*const PlTerm &t*)**

If *t* represents an atom, the new instance represents this atom. Otherwise a `type_error` is thrown.

#### **int PlAtom::operator==(const char \*text)**

Yields TRUE if the atom represents *text*, FALSE otherwise. Performs a `strcmp()` for this.

#### **int PlAtom::operator==(const PlAtom &a)**

Compares the two atom-handles, returning TRUE or FALSE.

## 7 The class PlRegister

This class encapsulates `PL_register_foreign()`. It is defined as a class rather than a function to exploit the C++ *global constructor* feature. This class provides a constructor to deal with the `PREDICATE()` way of defining foreign predicates as well as constructors to deal with more conventional foreign predicate definitions.

**PLRegister::PLRegister**(*const char \*module, const char \*name, int arity, foreign\_t (f)(term\_t t0, int a, control\_t ctx)*)

Register  $f$  as a the implementation of the foreign predicate  $\langle name \rangle / \langle arity \rangle$ . This interface uses the `PL_FA_VARARGS` calling convention, where the argument list of the predicate is passed using an array of `term_t` objects as returned by `PL_new_term_refs()`. This interface poses no limits on the arity of the predicate and is faster, especially for a large number of arguments.

**PLRegister::PLRegister**(*const char \*module, const char \*name, foreign\_t (\*f)(PlTerm a0, ...)*)

Registers functions for use with the traditional calling conventional, where each positional argument to the predicate is passed as an argument to the function  $f$ . This can be used to define functions as predicates similar to what is used in the C-interface:

```
static foreign_t
pl_hello(PlTerm a1)
{ ...
}

PLRegister x_hello_1(NULL, "hello", 1, pl_hello);
```

This construct is currently supported upto 3 arguments.

## 8 The class **PlQuery**

This class encapsulates the call-backs onto Prolog.

**PlQuery::PlQuery**(*const char \*name, const PlTermv &av*)

Create a query where  $name$  defines the name of the predicate and  $av$  the argument vector. The arity is deduced from  $av$ . The predicate is located in the Prolog module `user`.

**PlQuery::PlQuery**(*const char \*module, const char \*name, const PlTermv &av*)

Same, but performs the predicate lookup in the indicated module.

**int PlQuery::next\_solution()**

Provide the next solution to the query. Yields `TRUE` if successful and `FALSE` if there are no (more) solutions. Prolog exceptions are mapped to C++ exceptions.

Below is an example listing the currently defined Prolog modules to the terminal.

```
PREDICATE(list_modules, 0)
{ PlTermv av(1);

  PlQuery q("current_module", av);
  while( q.next_solution() )
    cout << (char *)av[0] << endl;

  return TRUE;
}
```

In addition to the above, the following functions have been defined.

```
int PlCall(const char *predicate, const PlTermv &av)
    Creates a PlQuery from the arguments generates the first next_solution() and destroys the
    query. Returns the result of next_solution() or an exception.

int PlCall(const char *module, const char *predicate, const PlTermv &av)
    Same, locating the predicate in the named module.

int PlCall(const char *goal)
    Translates goal into a term and calls this term as the other PlCall() variations. Especially suitable
    for simple goals such as making Prolog load a file.
```

## 8.1 The class PlFrame

The class PlFrame provides an interface to discard unused term-references as well as rewinding unifications (*data-backtracking*). Reclaiming unused term-references is automatically performed after a call to a C++-defined predicate has finished and returns control to Prolog. In this scenario PlFrame is rarely of any use. This class comes into play if the toplevel program is defined in C++ and calls Prolog multiple times. Setting up arguments to a query requires term-references and using PlFrame is the only way to reclaim them.

### **PlFrame::PlFrame()**

Creating an instance of this class marks all term-references created afterwards to be valid only in the scope of this instance.

### **PlFrame::~~PlFrame()**

Reclaims all term-references created after constructing the instance.

### **void PlFrame::rewind()**

Discards all term-references **and** global-stack data created as well as undoing all unifications after the instance was created.

A typical use for PlFrame is the definition of C++ functions that call Prolog and may be called repeatedly from C++. Consider the definition of assertWord(), adding a fact to word/1:

```
void
assertWord(const char *word)
{ PlFrame fr;
  PlTermv av(1);

  av[0] = PlCompound("word", PlTermv(word));
  PlQuery q("assert", av);
  q.next_solution();
}
```

This example shows the most sensible use of PlFrame if it is used in the context of a foreign predicate. The predicate's thruth-value is the same as for the Prolog unification ( $=/2$ ), but has no side effects. In Prolog one would use double negation to achieve this.



```

PREDICATE(can_unify, 2)
{ PlFrame fr;

    int rval = (A1=A2);
    fr.rewind();
    return rval;
}

```

## 9 The PREDICATE macro

The PREDICATE macro is there to make your code look nice, taking care of the interface to the C-defined SWI-Prolog kernel as well as mapping exceptions. Using the macro

```
PREDICATE(hello, 1)
```

is the same as writing:

```

static foreign_t pl_hello__1(PlTermv _av);

static foreign_t
_pl_hello__1(term_t t0, int arity, control_t ctx)
{ try
    { return pl_hello__1(PlTermv(1, t0));
    } catch ( PlTerm &ex )
    { return ex.raise();
    }
}

static PlRegister _x_hello__1("hello", 1, _pl_hello__1);

static foreign_t
pl_hello__1(PlTermv _av)

```

The first function converts the parameters passed from the Prolog kernel to a `PlTermv` instance and maps exceptions raised in the body to Prolog exceptions. The `PlRegister` global constructor registers the predicate. Finally, the function header for the implementation is created.

### 9.1 Controlling the Prolog destination module

With no special precautions, the predicates are defined into the module from which `load_foreign_library/1` was called, or in the module `user` if there is no Prolog context from which to deduce the module such as while linking the extension statically with the Prolog kernel.

Alternatively, *before* loading the SWI-Prolog include file, the macro `PROLOG_MODULE` may be defined to a string containing the name of the destination module. A module name may only contain alpha-numerical characters (letters, digits, \_). See the example below:

```

#define PROLOG_MODULE "math"
#include <SWI-Prolog.h>
#include <math.h>

PREDICATE(pi, 1)
{ A1 = M_PI;
}

?- math:pi(X).

X = 3.14159

```

## 10 Exceptions

Prolog exceptions are mapped to C++ exceptions using the subclass `PlException` of `PlTerm` to represent the Prolog exception term. All type-conversion functions of the interface raise Prolog-compliant exceptions, providing decent error-handling support at no extra work for the programmer.

For some commonly used exceptions, subclasses of `PlException` have been created to exploit both their constructors for easy creation of these exceptions as well as selective trapping in C++. Currently, these are `PlTypeError` and `PlDomainError`.

To throw an exception, create an instance of `PlException` and use `throw()` or `PlException::cppThrow()`. The latter refines the C++ exception class according to the represented Prolog exception before calling `throw()`.

```

char *data = "users";

throw PlException(PlCompound("no_database", PlTerm(data)));

```

### 10.1 The class `PlException`

This subclass of `PlTerm` is used to represent exceptions. Currently defined methods are:

#### **`PlException::PlException(const PlTerm &t)`**

Create an exception from a general Prolog term. This provides the interface for throwing any Prolog terms as an exception.

#### **`PlException::operator char *(void)`**

The exception is translated into a message as produced by `print_message/2`. The character data is stored in a ring. Example:

```

...;
try
{ PlCall("consult(load)");
} catch ( PlException &ex )
{ cerr << (char *) ex << endl;
}

```

int **plThrow()**

Used in the PREDICATE() wrapper to pass the exception to Prolog. See PL\_raise\_exception().

int **cppThrow()**

Used by PlQuery::next\_solution() to refine a generic PlException representing a specific class of Prolog exceptions to the corresponding C++ exception class and finally then executes throw(). Thus, if a PlException represents the term

error(type\_error(*Expected*, *Actual*), *Context*)

PlException::cppThrow() throws a PlTypeError exception. This ensures consistency in the exception-class whether the exception is generated by the C++-interface or returned by Prolog.

The following example illustrates this behaviour:

```
PREDICATE(call_atom, 1)
{ try
  { return PlCall((char *)A1);
  } catch ( PlTypeError &ex )
  { cerr << "Type Error caught in C++" << endl;
    cerr << "Message: \"" << (char *)ex << "\"\" << endl;
    return FALSE;
  }
}
```

## 10.2 The class PlTypeError

A *type error* expresses that a term does not satisfy the expected basic Prolog type.

**PlTypeError::PlTypeError(const char \*expected, const PlTerm &actual)**

Creates an ISO standard Prolog error term expressing the *expected* type and *actual* term that does not satisfy this type.

## 10.3 The class PlDomainError

A *domain error* expresses that a term satisfies the basic Prolog type expected, but is unacceptable to the restricted domain expected by some operation. For example, the standard Prolog open/3 call expect an io\_mode (read, write, append, ...). If an integer is provided, this is a *type error*, if an atom other than one of the defined io-modes is provided it is a *domain error*.

**PlDomainError::PlDomainError(const char \*expected, const PlTerm &actual)**

Creates an ISO standard Prolog error term expressing a the *expected* domain and the *actual* term found.

## 11 Embedded applications

Most of the above assumes Prolog is ‘in charge’ of the application and C++ is used to add functionality to Prolog, either for accessing external resources or for performance reasons. In some applications, there is a *main-program* and we want to use Prolog as a *logic server*. For these applications, the class `PlEngine` has been defined.

Only a single instance of this class can exist in a process. When used in a multi-threading application, only one thread at a time may have a running query on this engine. Applications should ensure this using proper locking techniques.<sup>1</sup>

**`PlEngine::PlEngine(int argc, char **argv)`**

Initialises the Prolog engine. The application should make sure to pass `argv[0]` from its main function, which is needed in the Unix version to find the running executable. See `PL_initialise()` for details.

**`PlEngine::PlEngine(char *argv0)`**

Simple constructure using the main constructor with the specified argument for `argv[0]`.

**`PlEngine::~PlEngine()`**

Calls `PL_cleanup()` to destroy all data created by the Prolog engine.

Section 4.11 has a simple example using this class.

## 12 Considerations

### 12.1 The C++ versus the C interface

Not all functionality of the C-interface is provided, but as `PlTerm` and `term_t` are essentially the same thing with automatic type-conversion between the two, this interface can be freely mixed with the functions defined for plain C.

Using this interface rather than the plain C-interface requires a little more resources. More term-references are wasted (but reclaimed on return to Prolog or using `PlFrame`). Use of some intermediate types (`functor_t` etc.) is not supported in the current interface, causing more hash-table lookups. This could be fixed, at the price of slightly complicating the interface.

### 12.2 Static linking and embedding

The mechanisms outlined in this document can be used for static linking with the SWI-Prolog kernel using `plld(1)`. In general the C++ linker should be used to deal with the C++ runtime libraries and global constructors. As of SWI-Prolog 3.2.9, `PL_register_foreign()` can be called *before* `PL_initialise()`, which is required to handle the calls from the global `PlRegister` calls.

---

<sup>1</sup>For Unix, there is a multi-threaded version of SWI-Prolog. In this version each thread can create and destroy a thread-engine. There is currently no C++ interface defined to access this functionality, though —of course— you can use the C-functions.

### 12.3 Status and compiler versions

The current interface is entirely defined in the `.h` file using inlined code. This approach has a few advantages: as no C++ code is in the Prolog kernel, different C++ compilers with different name-mangling schemas can cooperate smoothly.

Also, changes to the header file have no consequences to binary compatibility with the SWI-Prolog kernel. This makes it possible to have different versions of the header file with few compatibility consequences. If the interface stabilises we will consider options to share more code.

### 12.4 Limitations

Currently, the following limitations are recognised:

- *Predicate naming*  
Using the `PREDICATE()` macro, only predicates with a name that is valid as part of a C-symbol can be defined. Notably this makes the definition of predicates with names consisting of *symbol characters* impossible.
- *Non-deterministic predicates*  
The current interface does not provide for foreign-defined non-deterministic predicates. It would not be hard to add this.

## 13 Conclusions

In this document, we presented a high-level interface to Prolog employing automatic type-conversion and exception-handling defined in C++.

Programming using this interface is much more natural and requires only little extra resources in terms of time and memory.

Especially the smooth integration between C++ and Prolog exceptions reduce the coding effort for type checking and reporting in foreign predicates.

## Index

add/3, 6  
arg/3, 10  
assert, 16  
atom\_chars/2, 4, 11  
average/3, 6  
  
cppThrow(), 19  
  
entry/1, 11  
  
hello/1, 5  
  
load\_foreign\_library/1, 17  
  
open/3, 19  
  
PIAtom *class*, 5, 8, 13  
PIAtom::operator ==(), 14  
PIAtom::PIAtom(), 14  
PICall(), 16  
PICharList *class*, 4  
PICharList::PICharList(), 11  
PICodeList *class*, 4  
PICodeList::PICodeList(), 10  
PICompound *class*, 4, 10  
PICompound::PICompound(), 11  
PIDomainError *class*, 5, 18  
PIDomainError::PIDomainError(), 19  
PIEngine *class*, 5, 20  
PIEngine::~PIEngine(), 20  
PIEngine::PIEngine(), 20  
PIException *class*, 5, 18, 19  
PIException::operator char \*(), 18  
PIException::PIException(), 18  
PIFrame *class*, 5, 16, 20  
PIFrame::~PIFrame(), 16  
PIFrame::PIFrame(), 16  
PIFrame::rewind(), 16  
PIQuery *class*, 5, 6, 12, 16  
PIQuery::next\_solution(), 15  
PIQuery::PIQuery(), 15  
PIRegister *class*, 5, 17, 20  
PIRegister::PIRegister(), 15  
PIString *class*, 4  
PIString::PIString(), 10  
PITail *class*, 4, 11, 12  
  
PITail::append(), 11  
PITail::close(), 12  
PITail::next(), 12  
PITail::PITail(), 11  
PITerm *class*, 4–10, 18, 20  
PITerm::arity(), 10  
PITerm::name(), 10  
PITerm::operator !=(), 9  
PITerm::operator <(), 9  
PITerm::operator <=(), 9  
PITerm::operator >(), 9  
PITerm::operator >=(), 9  
PITerm::operator ==(), 8  
PITerm::operator ==(), 9  
PITerm::operator [](), 10  
PITerm::operator char \*(), 8  
PITerm::operator double(), 8  
PITerm::operator int(), 8  
PITerm::operator long(), 8  
PITerm::operator term\_t(), 8  
PITerm::operator void \*(), 8  
PITerm::PITerm(), 7  
PITerm::type(), 10  
PITermv *class*, 5, 11, 12, 17  
PITermv::PITermv(), 12  
plThrow(), 19  
PITypeError *class*, 18, 19  
PITypeError *class*, 5  
PITypeError::PITypeError(), 19  
print\_message/2, 18  
  
read/1, 11  
  
word/1, 16  
write/1, 6, 8