Writing C++ clients

polymake–workshop
Darmstadt 2011

Benjamin Lorenz

Goethe–Universität Frankfurt
blorenz@math.uni-frankfurt.de

01.04.2011

Basic client outline

```
#include "polymake/client.h"
#include "all you need from PTL"
#include "shared client headers"

namespace polymake { namespace APPNAME {

namespace { helper functions() {} }

type myfunction(perl::Object p, int x, perl::OptionSet opt)
{
}

Function4Perl(&myfunction,
"myfunction(Polytope ; $=1 , { opt => "none" } )");

} }
```

perl::Object

Constructors

`perl::Object p("type"[,"name"])`	open for new properties
`perl::Object p(q.type)`	open too
`perl::Object p=q.copy()`	immutable
`perl::Object p("newtype",q)`	immuable copy with a different type

perl::Object

Constructors

`perl::Object p("type"[,"name"])`	open for new properties
`perl::Object p(q.type)`	open too
`perl::Object p=q.copy()`	immutable
`perl::Object p("newtype",q)`	immuable copy with a different type

Properties

`p.take("NAME") << a`	if mutable or in prod. rule
`p.give("NAME") >> a`	assignment also possible
`p.exists("NAME")`	testing
`if(p.lookup("NAME") >> a)`	optional

Connecting to perl

There are four macros to connect your client to the perl world: `Function4perl`, `FunctionTemplate4perl`, `UserFunction4perl`, and `UserFunctionTemplate4perl`.

Connecting to perl

There are four macros to connect your client to the perl world: `Function4perl`, `FunctionTemplate4perl`, `UserFunction4perl`, and `UserFunctionTemplate4perl`.

- The `User...` variants take a description of the client as first argument. These also appear in TAB–completion.

Connecting to perl

There are four macros to connect your client to the perl world: `Function4perl`, `FunctionTemplate4perl`, `UserFunction4perl`, and `UserFunctionTemplate4perl`.

- The `User...` variants take a description of the client as first argument. These also appear in TAB–completion.
- The non `Template` variants take the address of the C function as next argument (`&myfunction`).

Connecting to perl

There are four macros to connect your client to the perl world: `Function4perl`, `FunctionTemplate4perl`, `UserFunction4perl`, and `UserFunctionTemplate4perl`.

- The `User...` variants take a description of the client as first argument. These also appear in TAB–completion.
- The non `Template` variants take the address of the C function as next argument (`&myfunction`).
- The last argument is a signature describing the perl function:

```
name < template_arg, ... > ( arg, ... ;
    opt_arg=default_value, ...
    { option_key => default_value , ... } )
    : attribute
```

Argument types

- $ for any number, string, or object

Argument types

- $ for any number, string, or object
- `Typename` for a perl–side name of the expected type

Argument types

- $ for any number, string, or object
- `Typename` for a perl–side name of the expected type
- `Typename<Typeparam,...>` for a concrete instance of a parameterized type (if each Typeparam is a concrete class), or an arbitrary instance (if any Typeparam is a placeholder introduced as template_arg.

Argument types

- $ for any number, string, or object
- `Typename` for a perl–side name of the expected type
- `Typename<Typeparam,...>` for a concrete instance of a parameterized type (if each Typeparam is a concrete class), or an arbitrary instance (if any Typeparam is a placeholder introduced as template_arg.
- $*$ for any object whose type can be recognized on the perl side

Calling polymake functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

Calling `polymake` functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

- Also neccesary to call scripts: `("script","name",par,...)`

Calling polymake functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

- Also neccesary to call scripts: `("script","name",par,...)`
- Similar functions exist to call user methods: `obj.CallPolymakeMethod()`

Calling polymake functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

- Also neccesary to call scripts: `("script","name",par,...)`
- Similar functions exist to call user methods: `obj.CallPolymakeMethod()`
- Arbitrary lists can be passed with `perl::ArgList`.

Calling polymake functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

- Also neccesary to call scripts: `("script","name",par,...)`
- Similar functions exist to call user methods: `obj.CallPolymakeMethod()`
- Arbitrary lists can be passed with `perl::ArgList`.
- `perl::Hash` for options in hash-style, or inline via `PolymakeOptions("name",val,"name",val,...)`

Calling polymake functions

```
VoidCallPolymakeFunction("name",par,...);
[a = ] CallPolymakeFunction("name",par,...) [ >> a];
perl::ListReturn l = ListCallPolymakeFunction("name",par,...);
ListCallPolymakeFunction("name",par,...) >> a >> b >> c;
```

- Also neccesary to call scripts: `("script","name",par,...)`
- Similar functions exist to call user methods: `obj.CallPolymakeMethod()`
- Arbitrary lists can be passed with `perl::ArgList`.
- `perl::Hash` for options in hash-style, or inline via `PolymakeOptions("name",val,"name",val,...)`
- Since perl is rather flexible you can call any function with any call, the result is usually what you would expect. But calling a list function in scalar context gives the *last* element.

Debugging your client

- Build a debug version of `polymake` with `make Debug=y`

Debugging your client

- Build a debug version of `polymake` with `make Debug=y`
- Set the following two options for `gdb` in `~/.gdbinit`

```
set breakpoint pending on
set env POLYMAKE_DEBUG_CLIENTS=-d
```

Debugging your client

- Build a debug version of `polymake` with `make Debug=y`
- Set the following two options for `gdb` in `~/.gdbinit`

```
set breakpoint pending on
set env POLYMAKE_DEBUG_CLIENTS=-d
```

- Run `gdb -args perl path/to/polymake -d`

Debugging your client

- Build a debug version of `polymake` with `make Debug=y`
- Set the following two options for gdb in `~/.gdbinit`

```
set breakpoint pending on
set env POLYMAKE_DEBUG_CLIENTS=-d
```

- Run `gdb -args perl path/to/polymake -d`
- Set a breakpoint in your client with `b client.cc:123`