Polymorphic Types

Introduction

Polymorphic Types are opaque data types that are designed as generic containers of data that can be safely passed around between blocks and threads in GNU Radio. They are heavily used in the stream tags and message passing interfaces. The most complete list of PMT function is, of course, the source code, specifically the header file **pmt.h**. This manual page summarizes the most important features and points of PMTs.

Let's dive straight into some Python code and see how we can use PMTs:

```
>>> import pmt
>>> P = pmt.from_long(23)
>>> type(P)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P
23
>>> P2 = pmt.from_complex(1j)
>>> type(P2)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P2
0+1i
>>> pmt.is_complex(P2)
True
```

First, the pmt module is imported. We assign two values (P and P2) with PMTs using the **from_long()** and **from_complex()** calls, respectively. As we can see, they are both of the same type! This means we can pass these variables to C++ through SWIG, and C++ can handle this type accordingly.

The same code as above in C++ would look like this:

```
#include <pmt/pmt.h>
// [...]
pmt::pmt_t P = pmt::from_long(23);
std::cout << P << std::endl;
pmt::pmt_t P2 = pmt::from_complex(gr_complex(0, 1)); // Alternatively: pmt::from_complex(0, 1)
std::cout << P2 << std::endl;
std::cout << pmt::is_complex(P2) << std::endl;
```

Two things stand out in both Python and C++: First we can simply print the contents of a PMT. How is this possible? Well, the PMTs have in-built capability to cast their value to a string (this is not possible with all types, though). Second, PMTs must obviously know their type, so we can query that, e.g. by calling the **is_complex()** method.

When assigning a non-PMT value to a PMT, we can use the from_* methods, and use the to_* methods to convert back:

```
pmt::pmt_t P_int = pmt::from_long(42);
int i = pmt::to_long(P_int);
pmt::pmt_t P_double = pmt::from_double(0.2);
double d = pmt::to_double(P_double);
```

String types play a bit of a special role in PMTs, as we will see later, and have their own converter:

```
pmt::pmt_t P_str = pmt::string_to_symbol("spam");
pmt::pmt_t P_str2 = pmt::intern("spam");
std::string str = pmt::symbol_to_string(P_str);
```

The **pmt::intern** is another way of saying **pmt::string_to_symbol**.

In Python, we can make use of the weak typing, and there's actually a helper function to do these conversions (C++ also has a helper function for converting to PMTs called **pmt::mp()**, but its less powerful, and not quite as useful, because types are always strictly known in C++):

```
P_int = pmt.to_pmt(42)
i = pmt.to_python(P_int)
P_double = pmt.to_pmt(0.2)
d = pmt.to_double(P_double)
```

On a side note, there are three useful PMT constants, which can be used in both Python and C++ domains. In C++, these can be used as such:

```
pmt::pmt_t P_true = pmt::PMT_T;
pmt::pmt_t P_false = pmt::PMT_F;
pmt::pmt_t P_nil = pmt::PMT_NIL;
```

In Python:

```
P_true = pmt.PMT_T
P_false = pmt.PMT_F
P_nil = pmt.PMT_NIL
```

pmt.PMT_T and pmt.PMT_F are boolean PMT types.

To be able to go back to C++ data types, we need to be able to find out the type from a PMT. The family of is_* methods helps us do that:

```
double d;
if (pmt::is_integer(P)) {
 d = (double) pmt::to_long(P);
} else if (pmt::is_real(P)) {
```

```
 d = pmt::to_double(P);
} else {
 // We really expected an integer or a double here, so we don't know what to do
 throw std::runtime_error("expected an integer!");
}
```

It is important to do type checking since we cannot unpack a PMT of the wrong data type.

We can compare PMTs without knowing their type by using the **pmt::equal()** function:

```
if (pmt::eq(P_int, P_double)) {
 std::cout << "Equal!" << std::endl; // This line will never be reached
```

The rest of this page provides more depth into how to handle different data types with the PMT library.

PMT Data Type

All PMTs are of the type **pmt::pmt_t**. This is an opaque container and PMT functions must be used to manipulate and even do things like compare PMTs. PMTs are also *immutable* (except PMT vectors). We never change the data in a PMT; instead, we create a new PMT with the new data. The main reason for this is thread safety. We can pass PMTs as tags and messages between blocks and each receives its own copy that we can read from. However, we can never write to this object, and so if multiple blocks have a reference to the same PMT, there is no possibility of thread-safety issues of one reading the PMT data while another is writing the data. If a block is trying to write new data to a PMT, it actually creates a new PMT to put the data into. Thus we allow easy access to data in the PMT format without worrying about mutex locking and unlocking while manipulating them.

PMTs can represent the following:

- Boolean values of true/false
- Strings (as symbols)
- Integers (long and uint64)
- Floats (as doubles)
- Complex (as two doubles)
- Pairs
- Tuples
- Vectors (of PMTs)
- Uniform vectors (of any standard data type)
- Dictionaries (list of key:value pairs)
- Any (contains a boost::any pointer to hold anything)

The PMT library also defines a set of functions that operate directly on PMTs such as:

- Equal/equivalence between PMTs
- Length (of a tuple or vector)
- Map (apply a function to all elements in the PMT)
- Reverse
- Get a PMT at a position in a list
- Serialize and deserialize
- Printing

The constants in the PMT library are:

- pmt::PMT_T - a PMT True
- pmt::PMT_F - a PMT False
- pmt::PMT_NIL - an empty PMT (think Python's 'None')

Inserting and Extracting Data

Use **pmt.h** for a complete guide to the list of functions used to create PMTs and get the data from a PMT. When using these functions, remember that while PMTs are opaque and designed to hold any data, the data underneath is still a C++ typed object, and so the right type of set/get function must be used for the data type.

Typically, a PMT object can be made from a scalar item using a call like "pmt::from_<type>". Similarly, when getting data out of a PMT, we use a call like "pmt::to_<type>". For example:

```
double a = 1.2345;
pmt::pmt_t pmt_a = pmt::from_double(a);
double b = pmt::to_double(pmt_a);

int c = 12345;
pmt::pmt_t pmt_c = pmt::from_long(c);
int d = pmt::to_long(pmt_c);
```

As a side-note, making a PMT from a complex number is not obvious:

```
std::complex<double> a(1.2, 3.4);
pmt::pmt_t pmt_a = pmt::make_rectangular(a.real(), b.imag());
std::complex<double> b = pmt::to_complex(pmt_a);
```

Pairs, dictionaries, and vectors have different constructors and ways to manipulate them, and these are explained in their own sections.

Strings

PMTs have a way of representing short strings. These strings are actually stored as interned symbols in a hash table, so in other words, only one PMT object for a given string exists. If creating a new symbol from a string, if that string already exists in the hash table, the constructor will return a reference to the existing PMT.

We create strings with the following functions, where the second function, **pmt::intern**, is simply an alias of the first.

```
pmt::pmt_t str0 = pmt::string_to_symbol(std::string("some string"));
pmt::pmt_t str1 = pmt::intern(std::string("some string"));
```

The string can be retrieved using the inverse function:

```
std::string s = pmt::symbol_to_string(str0);
```

Tests and Comparisons

The PMT library comes with a number of functions to test and compare PMT objects. In general, for any PMT data type, there is an equivalent "pmt::is_<type>". We can use these to test the PMT before trying to access the data inside. Expanding our examples above, we have:

```
pmt::pmt_t str0 = pmt::string_to_symbol(std::string("some string"));
if(pmt::is_symbol(str0))
 std::string s = pmt::symbol_to_string(str0);

double a = 1.2345;
pmt::pmt_t pmt_a = pmt::from_double(a);
if(pmt::is_double(pmt_a))
 double b = pmt::to_double(pmt_a);

int c = 12345;
pmt::pmt_t pmt_c = pmt::from_long(c);
if(pmt::is_long(pmt_a))
 int d = pmt::to_long(pmt_c);

\\ This will fail the test. Otherwise, trying to coerce \b pmt_c as a
\\ double when internally it is a long will result in an exception.
if(pmt::is_double(pmt_a))
 double d = pmt::to_double(pmt_c);
```

Dictionaries

PMT dictionaries and lists of key:value pairs. They have a well-defined interface for creating, adding, removing, and accessing items in the dictionary. Note that every operation that changes the dictionary both takes a PMT dictionary as an argument and returns a PMT dictionary. The dictionary used as an input is not changed and the returned dictionary is a new PMT with the changes made there.

The following is a list of PMT dictionary functions. Click through to get more information on what each does.

- bool **pmt::is_dict(const pmt_t &obj)**
- pmt_t **pmt::make_dict()**
- pmt_t **pmt::dict_add(const pmt_t &dict, const pmt_t &key, const pmt_t &value)**
- pmt_t **pmt::dict_delete(const pmt_t &dict, const pmt_t &key)**
- bool **pmt::dict_has_key(const pmt_t &dict, const pmt_t &key)**
- pmt_t **pmt::dict_ref(const pmt_t &dict, const pmt_t &key, const pmt_t &not_found)**
- pmt_t **pmt::dict_items(pmt_t dict)**
- pmt_t **pmt::dict_keys(pmt_t dict)**
- pmt_t **pmt::dict_values(pmt_t dict)**

This example does some basic manipulations of PMT dictionaries in Python. Notice that we pass the dictionary *a* and return the results to *a*. This still creates a new dictionary and removes the local reference to the old dictionary. This just keeps our number of variables small.

```
import pmt

key0 = pmt.intern("int")
val0 = pmt.from_long(123)
val1 = pmt.from_long(234)

key1 = pmt.intern("double")
val2 = pmt.from_double(5.4321)

# Make an empty dictionary
a = pmt.make_dict()

# Add a key:value pair to the dictionary
a = pmt.dict_add(a, key0, val0)
print a

# Add a new value to the same key;
# new dict will still have one item with new value
a = pmt.dict_add(a, key0, val1)
print a

# Add a new key:value pair
a = pmt.dict_add(a, key1, val2)
print a
```

```
# Test if we have a key, then delete it
print pmt.dict_has_key(a, key1)
a = pmt.dict_delete(a, key1)
print pmt.dict_has_key(a, key1)

ref = pmt.dict_ref(a, key0, pmt.PMT_NIL)
print ref

# The following should never print
if(pmt.dict_has_key(a, key0) and pmt.eq(ref, pmt.PMT_NIL)):
 print "Trouble! We have key0, but it returned PMT_NIL"
```

Vectors

PMT vectors come in two forms: vectors of PMTs and vectors of uniform data. The standard PMT vector is a vector of PMTs, and each PMT can be of any internal type. On the other hand, uniform PMTs are of a specific data type which come in the form:

- (u)int8
- (u)int16
- (u)int32
- (u)int64
- float32
- float64
- complex 32 (std::complex<float>)
- complex 64 (std::complex<double>)

That is, the standard sizes of integers, floats, and complex types of both signed and unsigned.

Vectors have a well-defined interface that allows us to make, set, get, and fill them. We can also get the length of a vector with **pmt::length**.

For standard vectors, these functions look like:

- bool **pmt::is_vector(pmt_t x)**
- pmt_t **pmt::make_vector(size_t k, pmt_t fill)**
- pmt_t **pmt::vector_ref(pmt_t vector, size_t k)**
- void **pmt::vector_set(pmt_t vector, size_t k, pmt_t obj)**
- void **pmt::vector_fill(pmt_t vector, pmt_t fill)**

Uniform vectors have the same types of functions, but they are data type-dependent. The following list tries to explain them where you substitute the specific data type prefix for *dtype* (prefixes being: u8, u16, u32, u64, s8, s16, s32, s64, f32, f64, c32, c64).

- bool pmt::is_(dtype)vector(pmt_t x)
- pmt_t pmt::make_(dtype)vector(size_t k, (dtype) fill)
- pmt_t pmt::init_(dtype)vector(size_t k, const (dtype*) data)
- pmt_t pmt::init_(dtype)vector(size_t k, const std::vector<dtype> data)
- pmt_t pmt::(dtype)vector_ref(pmt_t vector, size_t k)
- void pmt::(dtype)vector_set(pmt_t vector, size_t k, (dtype) x)
- const dtype* pmt::(dtype)vector_elements(pmt_t vector, size_t &len)
- dtype* pmt::(dtype)vector_writable_elements(pmt_t vector, size_t &len)

Note: We break the contract with vectors. The 'set' functions actually change the data underneath. It is important to keep track of the implications of setting a new value as well as accessing the 'vector_writable_elements' data. Since these are mostly standard data types, sets and gets are atomic, so it is unlikely to cause a great deal of harm. But it's only unlikely, not impossible. Best to use mutexes whenever manipulating data in a vector.

BLOB

A BLOB is a 'binary large object' type. In PMT's, this is actually just a thin wrapper around a u8vector.

Pairs

Pairs are inspired by LISP 'cons' data types, so you will find the language here comes from LISP. A pair is just a pair of PMT objects. They are manipulated using the following functions:

- bool **pmt::is_pair(const pmt_t &obj)**: Return true if obj is a pair, else false
- pmt_t **pmt::cons(const pmt_t &x, const pmt_t &y)**: construct new pair
- pmt_t **pmt::car(const pmt_t &pair)**: get the car of the pair (first object)
- pmt_t **pmt::cdr(const pmt_t &pair)**: get the cdr of the pair (second object)
- void **pmt::set_car(pmt_t pair, pmt_t value)**: Stores value in the car field
- void **pmt::set_cdr(pmt_t pair, pmt_t value)**: Stores value in the cdr field

Serializing and Deserializing

It is often important to hide the fact that we are working with PMTs to make them easier to transmit, store, write to file, etc. The PMT library has methods to serialize data into a string buffer or a string and then methods to deserialize the string buffer or string back into a PMT. We use this extensively in the metadata files (see **Metadata Information**).

- bool **pmt::serialize(pmt_t obj, std::streambuf &sink)**
- std::string **pmt::serialize_str(pmt_t obj)**
- pmt_t **pmt::deserialize(std::streambuf &source)**
- pmt_t **pmt::deserialize_str(std::string str)**

For example, we will serialize the data above to make it into a string ready to be written to a file and then deserialize it back to its original PMT.

```
import pmt

key0 = pmt.intern("int")
val0 = pmt.from_long(123)

key1 = pmt.intern("double")
val1 = pmt.from_double(5.4321)

# Make an empty dictionary
a = pmt.make_dict()

# Add a key:value pair to the dictionary
a = pmt.dict_add(a, key0, val0)
a = pmt.dict_add(a, key1, val1)

print a

ser_str = pmt.serialize_str(a)
print ser_str

b = pmt.deserialize_str(ser_str)
print b
```

The line where we 'print ser_str' will print and parts will be readable, but the point of serializing is not to make a human-readable string. This is only done here as a test.

Printing

In Python, the **repr** function of a PMT object is overloaded to call '**pmt::write_string**'. This means that any time we call a formatted printing operation on a PMT object, the PMT library will properly format the object for display.

In C++, we can use the 'pmt::print(object)' function or print the contents is using the overloaded "<<" operator with a stream buffer object. In C++, we can inline print the contents of a PMT like:

```
pmt::pmt_t a pmt::from_double(1.0);
std::cout << "The PMT a contains " << a << std::endl;
```

Conversion between Python Objects and PMTs

Although PMTs can be manipulated in Python using the Python versions of the C++ interfaces, there are some additional goodies that make it easier to work with PMTs in python. There are functions to automate the conversion between PMTs and Python types for booleans, strings, integers, longs, floats, complex numbers, dictionaries, lists, tuples and combinations thereof.

Two functions capture most of this functionality:

```
pmt.to_pmt # Converts a python object to a PMT.
pmt.to_python # Converts a PMT into a python object.
```