Py++ Documentation

Release 1.9.0

Roman Yakovenko

Nov 02, 2017

Contents

Py++ is an object-oriented framework for creating a code generator for Boost.Python library and ctypes package.

Py++ uses few different programming paradigms to help you to expose C++ declarations to Python. This code generator will not stand on your way. It will guide you through the whole process. It will raise warnings in the case you are doing something wrong with a link to the explanation. And the most important it will save your time - you will not have to update code generator script every time source code is changed.

CHAPTER 1

Code generation process

Code generation process, using *Py++* consists from few simple steps.

1.1 *"read declarations"*

Py++ does not reinvent the wheel. It uses Clang C++ compiler to parse C++ source files. To be more precise, the tool chain looks like this:

1. source code is passed to CastXML
2. CastXML passes it to Clang C++ compiler
3. CastXML generates an XML description of a C++ program from Clang's internal representation.
4. *Py++* uses pygccxml package to read CastXML generated file.

The bottom line - you can be sure, that all your declarations are read correctly.

1.2 *"build module"*

Only very small and simple projects could be exported as is. Most of the projects still require human invocation. Basically there are 2 questions that you should answer:

1. which declarations should be exported
2. how this specific declaration should be exported or, if I change a little a question, what code should be written in order I get access from Python to that functionality

Of course, *Py++* cannot answer those question, but it provides maximum help to you.

How can *Py++* help you with first question? *Py++* provides very powerful and simple query interface. For example in one line of code you can select all free functions that have two arguments, where the first argument has type `int &` and the type of the second argument is any:

```
mb = module_builder_t( ... ) #module_builder_t is the main class that
                             #will help you with code generation process
mb.free_functions( arg_types=[ 'int &', None ] )
```

Another example - the developer wants to exclude all protected functions from being exported:

```
mb = module_builder_t( ... )
mb.calldefs( access_type_matcher_t( 'protected' ) ).exclude()
```

The developer can create custom criteria, for example exclude all declarations with 'impl' (implementation) string within the name.

```
mb = module_builder_t( ... )
mb.decls( lambda decl: 'impl' in decl.name ).exclude()
```

Note the way the queries were built. You can think about those queries as the rules, which will continue to work even after exported C++ code was changed. It means that you don't have to change code generator source code every time.

So far, so good what about second question? Well, by default *Py++* generates a code that will satisfy almost all developers. *Py++* could be configured in many ways to satisfy your needs. But sometimes this is still not enough. There are use cases when you need full control over generated code. One of the biggest problems, with code generators in general, is modifying generated code and preserving changes. How many code generators did you use or know, that allow you to put your code anywhere or to reorder generated code as you wish? *Py++* allows you to do that.

Py++ introduces new concept: code creator and code creators tree. You can think about code creators tree as some kind of AST. The only difference is that code creators tree provides more specific functionality. For example `include_t` code creator is responsible to create C++ `include` directive code. You have full control over code creators tree, before it is written to disc. Here you can find UML diagram of almost all code creators: class diagram.

At the end of this step you have code creators tree, which is ready to be written to disc.

1.3 *"write code to files"*

During this step *Py++* reads the code creators tree and writes the code to a disc. The code generation process result should not be different from the one, which would be achieved by human. For small project writing all code into single file is good approach, for big ones the code should be split into multiple files. *Py++* implements both strategies.

CHAPTER 2

Features list

- *Py++* supports almost all features found in Boost.Python library
- Using *Py++* you can develop few extension modules simultaneously, especially when they share the code.
- ***Py++* generates code, which will help you:**
  - to understand compiler generated error messages
  - to minimize project built time
- *Py++* has few modes of writing code into files: * single file * multiple files * fixed set of multiple files * multiple files, where single class code is split to few files
- You have full control over generated code. Your code could be inserted almost anywhere.
- Your license is written at the top of every generated file
- *Py++* will check the "completeness" of the bindings. It will check for you that the exposed declarations don't have references to unexposed ones.
- *Py++* provides enough functionality to extract source code documentation and write it as Python documentation string
- *Py++* provides simple and powerful framework to create a wrapper for functions, which could not be exposed as is to Python.

CHAPTER 3

License

Boost Software License.

CHAPTER 4

Documentation contents

4.1 Tutorials

4.1.1 Tutorials

Preamble

I guess you decided to try *Py++* API. Good! Lets start. First of all, please take a look on two files:

- *hello_world.hpp* - C++ source code, that we want to export to Python
- *generate_code.py* - Python code, that uses *Py++* to export declarations from the source file

module_builder_t

Py++ is built from a few packages, but there is only one package, you should really to be familiar with - `module_builder`. This package is some kind of facade to low level API. It provides simple and intuitive API. The main class within this package is `module_builder_t`. The instance of this class will guide you through the whole process.

module_builder_t.__init__

First of all, what is needed in order to create an instance of the class?

`module_builder_t.__init__` methods take few arguments:

1. `files` - list of all C++ source files, that declarations from them, you want to expose. This is the only required parameter.
2. `gccxml_path` - path to CastXML binary. If you don't supply this argument pygccxml will look for it using `PATH` environment variable.

There are some other arguments:

- additional include directories
- [un]defined symbols (macros)
- intermediate results cache strategy configuration
- ...

Parsing of source files is done within this method. Post condition of this method is that all declarations has been successfully extracted from the sources files and you can start work on them.

Declarations customization

There are always declarations, which should not be exported or could not be exported without human invocation. As you already saw from example, *Py++* provides simple and powerful declarations query interface. By default, only the declarations that belongs to the files, you have asked to parse, and to the files, that lies in the same directories as parsed files, will be exported:

```
project_root/
    details/
        impl_a.h
        impl_b.h
  include/
      a.h   //includes impl_a.h
      b.h   //includes impl_b.h
      all.h //includes a.h and b.h
mb = module_builder( [ 'all.h' ] )
```

All declarations that belongs to `include` directory will be signed as included to exporting. All other declarations will be ignored.

You can change the set of exported declarations by calling `exclude` or `include` methods, on declarations.

Basically, this is a second step of code generation process. During this step you could/should/may change *Py++* defaults:

- to rename exposed declarations
- to include/exclude declarations
- to set call policies
- ...

I think it is critical for beginners to see what is going on, right? `module_builder_t` class has `print_declarations` method. You can print whole declarations tree or some specific declaration. You will find a lot of useful information there:

- whether the declaration is include\excluded
- call policies
- warnings, *Py++* warns you about the declarations that have some “problems”
- ...

Very convenient, very useful.

module_builder_t.build_code_creator

Now it is a time to create module code creator. Do you remember, in introduction to *Py++*, I told you that before writing code to disc, *Py++* will create some kind of AST. Well calling module_builder_t.build_code_creator function does this. Right now, the only important argument to the function is `module_name`. Self explained, is it?

What is the main value of code creators? Code creators allow you to modify code before it has been written to disk. For example one common requirement for open source projects it to have license text within every source file. You can do it with one line of code only:

```
mb.code_creator.license = <<<your license text>>>
```

After you call this function, I recommend you not to change declarations configuration. In most cases the change will take effect, in some cases it will not!

This tutorial is not cover code creators and how you should work with them. I will write another tutorial.

module_builder_t.write_module

You have almost created your module. The last things left is to write module code to file(s). You can do it with

- `module_builder_t.write_module` - you should provide file name, the code will be written in.
- `module_builder_t.split_module` - you should provide directory name. For big projects it is a must to minimize compilation time. So *Py++* splits your module source code to different files within the directory.

Files

C++ header file

```
// Copyright 2004-2006 Roman Yakovenko.
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#ifndef __hello_world_hpp__
#define __hello_world_hpp__

#include <string>

//I want to rename color to Color
enum color{ red, green, blue };

struct genealogical_tree{/*...*/};

struct animal{

    explicit animal( const std::string& name="" )
    : m_name( name )
    {}

    //I need to set call policies to the function
    genealogical_tree& genealogical_tree_ref()
    { return m_genealogical_tree; }

    std::string name() const
```

```
    { return m_name; }

private:
    std::string m_name;
    genealogical_tree m_genealogical_tree;
};

//I want to exclude next declarations:
struct impl1{};
struct impl2{};

inline int* get_int_ptr(){ return 0;}
inline int* get_int_ptr(int){ return 0;}
inline int* get_int_ptr(double){ return 0;}

#endif//__hello_world_hpp__
```

module_builder_t usage example

```
#! /usr/bin/python
# Copyright 2004-2008 Roman Yakovenko.
# Distributed under the Boost Software License, Version 1.0. (See
# accompanying file LICENSE_1_0.txt or copy at
# http://www.boost.org/LICENSE_1_0.txt)

import os
import sys
sys.path.append( '../../../..' )
from environment import gccxml
from pyplusplus import module_builder

mb = module_builder.module_builder_t(
        files=['hello_world.hpp']
        , gccxml_path=gccxml.executable ) #path to gccxml executable

#rename enum Color to color
Color = mb.enum( 'color' )
Color.rename('Color')

#Set call policies to animal::genealogical_tree_ref
animal = mb.class_( 'animal' )
genealogical_tree_ref = animal.member_function( 'genealogical_tree_ref',␣
↪recursive=False )
genealogical_tree_ref.call_policies = module_builder.call_policies.return_internal_
↪reference()

#next code has same effect
genealogical_tree_ref = mb.member_function( 'genealogical_tree_ref' )
genealogical_tree_ref.call_policies = module_builder.call_policies.return_internal_
↪reference()

#I want to exclude all classes with name starts with impl
impl_classes = mb.classes( lambda decl: decl.name.startswith( 'impl' ) )
impl_classes.exclude()

#I want to exclude all functions that returns pointer to int
```

```
ptr_to_int = mb.free_functions( return_type='int *' )
ptr_to_int.exclude()

#I can print declarations to see what is going on
mb.print_declarations()

#I can print single declaration
mb.print_declarations( animal )

#Now it is the time to give a name to our module
mb.build_code_creator( module_name='hw' )

#It is common requirement in software world - each file should have license
mb.code_creator.license = '//Boost Software License( http://boost.org/more/license_
↪info.html )'

#I don't want absolute includes within code
mb.code_creator.user_defined_directories.append( os.path.abspath('.') )

#And finally we can write code to the disk
mb.write_module( os.path.join( os.path.abspath('.'), 'generated.cpp' ) )
```

Generated code

```
// This file has been generated by Py++.

//Boost Software License( http://boost.org/more/license_info.html )

#include "boost/python.hpp"

#include "hello_world.hpp"

namespace bp = boost::python;

BOOST_PYTHON_MODULE(hw){
    bp::enum_< color>("Color")
        .value("red", red)
        .value("green", green)
        .value("blue", blue)
        .export_values()
        ;

    bp::class_< animal, boost::noncopyable >( "animal", bp::init< bp::optional<␣
↪std::string const & > >(( bp::arg("name")="" )) )
        .def(
            "genealogical_tree_ref"
            , &::animal::genealogical_tree_ref
            , bp::return_internal_reference< 1, bp::default_call_policies >() )
        .def(
            "name"
            , &::animal::name );

    bp::implicitly_convertible< std::string const &, animal >();

    bp::class_< genealogical_tree >( "genealogical_tree" );
}
```

That's all. I hope you enjoyed.

4.1.2 Documentation string

Introduction

Py++ provides a convenient way to export documentation from C++ source files as Python documentation string

API description

```
mb = module_builder_t( ... )
my_class = mb.class_( 'my_class' )
my_class.documentation = '"very helpful documentation string"'
my_class.member_function( "do_nothing" ).documentation = \
    '"This function does nothing."'
```

In *Py++* every class, which describes C++ declarations has `documentation` property. This property should contain valid C++ string or `None`.

Boost.Python not always provides functionality, which exports documentation string. In those cases, *Py++* will not generate documentation string.

Also the previous method is pretty clear, it is not practical. There should be a better way, to complete the task. Lets take a look on `module_builder_t.build_code_creator` method. One of the arguments of this method is `doc_extractor`.

`doc_extractor` is a callable object, which takes one argument - reference to declaration.

```
def doc_extractor( decl ):
  ...
```

How it could help? Every declaration has location information:

- `decl.location.file_name` - absolute file name, where this declaration has been declared.
- `decl.location.line` - line number.

So, you can go to the source file and to extract declaration from it. *Py++* will call `doc_extractor` on every exported declaration.

Now, when I think you understand what functionality *Py++* provides. It is a time to say what functionality is missing. *Py++* does not provide any documentation extractor. It is not completely true. You can find document extractor for doxygen format in `contrib/doc_extractors` directory. Georgiy Dernovoy has contributed it.

4.1.3 Inserting code

Introduction

Py++ is not a magician! Sometimes there is a need to add code to generated file(s). This document will describe how you can insert your code to almost any place.

Insert code to module

Almost every `Boost.Python` module has the following structure:

```
//declarations code
...
BOOST_PYTHON_MODULE(X)
{
  //registrations code
  ...
}
```

Using `module_builder_t` you can add code to declaration and registration sections. More over you can add the code to head or tail of the section. `module_builder_t` class provides API, which will help you to complete the task:

- `add_declaration_code( self, code, tail=True )`

  This function will add a code to the declaration section. If you want to add the code to the head of the section, pass `tail=False` to the method.

- `add_registration_code( self, code, tail=True )`

  This function will add a code to the registration section. If you want to add the code to the head of the section, pass `tail=False` to the method.

Example

```
mb = module_builder_t( ... )
mb.build_code_creator( ... )
mb.add_declaration_code( '//just a comment' )
mb.add_registration_code( '//another comment', False ) #adding code to the head
```

Insert code to class

`class_t` declaration defines few methods, which add user code to the generated one. Lets take a look on the following use case:

```
struct window_t{
    ...
    void get_size( int& height, int& width ) const;
    ...
};
```

`int` is immutable type in Python. So you cannot expose `get_size` member function as is. You need to create a wrapper and expose it.

In the near future *Py++* will eliminate the need of creating hand written wrapper for this use case.

```
boost::python::tuple get_window_size( const window_t& win ){
    int h(0), w(0);
    win.get_size( h, w );
    return boost::python::make_tuple( h, w );
}
```

Now you have to register it:

```
using boost::python;
class_< window_t >( ... )
    .def( "get_size", &::get_window_size )
    ...
;
```

How it could be achieved with *Py++*? Class declaration, has also two functions:

- `add_declaration_code( self, code )`

  This method will add the code to the declaration section within the module.

  If you split your module to few files, *Py++* will add this code to the "cpp" file, class registration code will be written in.

  Attention: there is no defined order between wrapper code and declaration section code. If you have dependencies between code from declaration section and class wrapper, consider to move declaration code to class wrapper.

- `add_registration_code( self, code, works_on_instance=True )`

  This method will add the code to the registration section of the class.

  What is `works_on_instance` argument for? In our case, we added new method to the class. The first argument of the call will be `self`.

  ```
  #From Python user can call this method like this:
  win = window_t( )
  height, width = win.get_size()
  ```

  If you will pass `works_on_instance=False` the following code will be generated:

  ```
  {
      class_< window_t > window_exporter( "window_t" );
      scope window_scope( window_exporter );
      ...
      def( "get_size", &::get_window_size );
  }
  ```

  And in this case, user will be forced to pass reference to `window_t` object:

  ```
  win = window_t()
  height, width = window_t.get_size( win )
  ```

Example

```
mb = module_builder_t( ... )
window = mb.class_( 'window_t' )
window.add_declaration_code( get_window_size definition )
window.add_registration_code( 'def( "get_size", &::get_window_size )' )
#Py++ will add ';' if needed
```

Insert code to class wrapper

There are use cases, when you have to add code to class wrapper. Please take a look on the following thread: http://mail.python.org/pipermail/c++-sig/2006-June/010791.html .

The short description is the following: there are classes with parent/child relationship. Parent keeps child class instances using raw pointer. When parent die, it also destroys children classes. It is not an option to switch to `boost::shared_ptr`.

The solution Niall Douglas found was to implement small lifetime manager. For this solution he needed:

- to add to every constructor of class wrapper some code that registers the instance of the class within the manager
- to add to destructor of class wrapper some code, that will destroy the instance if needed.
- to add to class wrapper new variable

Solution

```
def inject_code( cls ):
    constructors = cls.constructors()
    constructors.body = class instance registration code
    #if you need to add code to default or copy constructor only
    #than you can you the following shortcuts
    cls.null_constructor_body = <<<your code>>>
    cls.copy_constructor_body = <<<your code>>>
    #which will update the appropriate ``body`` property.

    #If you want to add code to the class destructor,
    #use ``add_destructor_code`` method
    cls.add_destructor_code( <<<your code>>> )

    #If you need to add new class variables:
    cls.add_wrapper_code( <<<variable declaration>>> )
```

```
mb = module_builder_t( ... )
for cls in mb.classes( <<<relevant classes only>>> ):
  inject_code( cls )
```

Header files

Now, when you know how to add your code to a generated one, I think you also should now how to add your own set of include directives to the generated files. There are few ways to do this.

1. The easiest and the most effective one - tell to *Py++* that generated code for the declaration should include additional files:

   ```
   mb = module_builder_t( ... )
   my_class = mb.class_( ... )
   my_class.include_files.append( "vector" )
   ```

   Every declaration has `include_files` property. This is a list of header files, you want to include from the generated file(s).

2. Other approach is a little bit low level, but it allows you to add your header files to every generated file:

   ```
   mb = module_builder_t( ... )
   ...
   mb.build_code_creator( ... )
   mb.code_creator.add_include( "iostream" )
   ```

You can also replace all (to be) generated header files with your own set:

```
mb.code_creator.replace_included_headers( ["stdafx.h"] )
```

Of course you can, and may be should, use both approaches.

I suggest you to spend some time and to tweak *Py++* to generate source code with as little as possible include directives. This will save you huge amount of time later.

4.1.4 Multi-module development

Introduction

It is a common practices to construct final program or a package from few different dependent or independent C++ libraries. Many time these libraries reuse classes\functions defined in some other library. I think this is a must requirement from a code generator to be able to expose these libraries to Python , without "re-exposing" the class\functions definition twice.

This functionality is new in version "0.8.6".

Use case introduction

Lets say that you have to expose few libraries, which deal with image processing:

- `core` library - defines base class for all image classes - `image_i`
- `png` library - defines class `png_image_t`, which derives from `core::image_i` and implements functionality for "png" image format.

The code:

```
namespace core{
    class image_i{
        ...
        virtual void load() = 0;
    };
} //core

namespace png{
    class png_image_t : public core::image_i{
        ...
        virtual void load();
    };
}
```

The desired goal is to expose every class in its own package.

already_exposed

Every *Py++* declaration has `already_exposed` property. This property says to *Py++* that the declaration is already exposed in another module:

```
#generate_code.py script

mb_core = module_builder_t( ... )
mb_core.class_( 'image_i' ).include()
```

```
mb_core.build_code_creator( 'core' )
mb.write_module( 'core.cpp' )

mb_png = module_builder_t( ... )
mb_png.class_( '::core::image_i' ).already_exposed = True
mb_png.class_( '::png::png_image_t' ).include()
mb_core.build_code_creator( 'png' )
mb.write_module( 'png.cpp' )
```

Py++ will generate code very similar to the the following one:

```
//file core.cpp
namespace bp = boost::python;

struct image_i_wrapper : core::image_i, bp::wrapper< core::image_i > {
    image_i_wrapper()
    : core::image_i(), bp::wrapper< core::image_i >()
    {}

    virtual void load(  ){
        bp::override func_load = this->get_override( "load" );
        func_load(  );
    }
    ...
};

BOOST_PYTHON_MODULE(core){
    bp::class_< image_i_wrapper, boost::noncopyable >( "image_i" )
        ...
        .def( "load", bp::pure_virtual( &::core::image_i::load ) );
}
```

```
//file png.cpp
struct png_image_t_wrapper : png::png_image_t, bp::wrapper< png::png_image_t > {

    png_image_t_wrapper()
    : png::png_image_t(), bp::wrapper< png::png_image_t >()
    {}

    virtual void load(  ) {
        if( bp::override func_load = this->get_override( "load" ) )
            func_load(  );
        else
            this->png::png_image_t::load(  );
    }

    void default_load(  ) {
        png::png_image_t::load( );
    }
};

BOOST_PYTHON_MODULE(pyplusplus){
    bp::class_< png_image_t_wrapper, bp::bases< core::image_i > >( "png_image_t" )
    //---------------------------------^^^^^^^^^^^^^^^^^^^^^^^^^^
      ...
      .def( "load", &::png::png_image_t::load, &png_image_t_wrapper::default_load );
}
```

As you can see "png.cpp" file doesn't contains code, which exposes `core::image_i` class.

Semi-automatic solution

`already_exposed` solution is pretty good when you mix hand-written modules with the Py++ generated ones. It doesn't work/scale for "true" multi-module development. This is exactly the reason why *Py++* offers "semi automatic" solution.

For every exposed module, *Py++* generates "exposed_decl.pypp.txt" file. This file contains the list of all parsed declarations and whether they were included or excluded. Later, when you work on another module, you can tell *Py++* that the current module depends on the previously generated one. *Py++* will load "exposed_decl.pypp.txt" file and update the declarations.

Usage example:

```
mb = module_builder_t( ... )
mb.register_module_dependency( <<<other module generated code directory>>> )
```

Caveat

You should import module "core", before "png". Boost.Python requires definition of any base class to be exposed\registered before a derive one.

4.1.5 Properties

Introduction

Boost.Python allows users to specify class properties. You can read about this functionality in the tutorials or in the reference manual. Since version 0.8.2 *Py++* provides a convenient API to specify class properties.

Usage example

```
struct number{
    ...
    float value() const;
    void  set_value( float );
    ...
private:
    float m_value;
}
```

```
mb = module_builder_t( ... )
number = mb.class_( 'number' )
number.add_property( 'ro_value', number.member_function( 'value' ) )
number.add_property( 'value'
                     , number.member_function( 'value' )
                     , number.member_function( 'set_value' ) )
```

This is rather the hard way to add properties to the class. *Py++* comes with built-in algorithm, which automatically recognizes properties and adds them to the class:

```
mb = module_builder_t( ... )
number = mb.class_( 'number' )
number.add_properties( exclude_accessors=False ) #accessors will be exposed
```

Small advise to you: try `add_properties` algorithm first, it should work. If it doesn't than:

- Please, bring your use case to the developers of *Py++*, so we could improve it
- Switch to the `add_property` method for a while

Call policies

Consider the following use case:

```
struct nested{ ... };
```

```
struct data{
    ...
    const nested& get_nested() const
    { return m_nested; }
    ...
private:
    nested m_nested;
};
```

In order to expose `get_nested` member function you have to specify its call policies. Same precondition holds for exposing member function as property:

```
mb = module_builder_t( ... )
get_nested = mb.member_function( 'get_nested' )
get_nested.call_policies = call_policies.return_internal_reference()
mb.class_( 'data' ).add_properties()
```

Py++ will take the call policies information from the relevant accessor.

Property recognition algorithm

Description

In general the algorithm is very simple. *Py++* understands few coding conventions. It is aware of few widely used get\set prefixes. It scans the class and its base classes for accessors and after this it tries to match between "get" and "set" accessors. If there is "set" accessors, but there is no "get" accessor, property will not be constructed. At least one accessor should belong to the class. In new property will override an existing exposed declarations property will not be created and warning will be written.

Find accessors

This part of the algorithm is responsible for finding all functions, which meet get\set accessors criteria.

"get" accessor criteria

1. It does not have arguments.

2. It has return other than `void`.
3. It does not modify the instance - has `const` attribute.
4. It does not have an overload.

"set" accessor criteria

1. It has only 1 argument.
2. Its return type is `void`.
3. It do modify the instance - doesn't have `const` attribute.

There are also few rules that applies on both accessor types:

1. Accessor should be **included**.
2. Accessor should be "public".
3. It should not be static.
4. It should not be pure virtual.

Recognize property

This part of the algorithm is responsible to recognize the pair of "get" and "set" accessors, which constructs the property. *Py++* does it by analyzing name and type of the accessors.

Py++ understands the following coding conventions:

- lowercase_with_underscores
- UpperCamel
- lowCamel

It is also aware of few common prefixes for set\get accessors: get, is, has, set, <<empty prefix for get accessor>>.

Documentation

You can use `doc` attribute to specify the property documentation. If you don't, than *Py++* will construct documentation, which will describe from what functions this property was built from.

4.1.6 Splitting generated code to files

Introduction

Py++ provides 4 different strategies for splitting the generated code into files:

- single file
- multiple files
- fixed set of multiple files
- multiple files, where single class code is split to few files

Single file

If you just start with *Py++* or you are developing small module, than you should start with this strategy. It is simple - all source code generated to a single file.

Of course this solution has it's price - every time you change the code you will have to recompile it. If you expose 2 or more declarations, this is annoying and time-consuming operation. In some cases you even will not be able to compile the generated code, because of its size.

Usage example

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t(...)
mb.build_code_creator( ... )
mb.write_module( <<<file name>>> )
```

Multiple files

I believe this is the most widely used strategy. *Py++* splits generated code as follows:

- every class has it's own source & header file
- the following declarations are split to separate source files:
  - named & unnamed enumerations
  - free functions
  - global variables
- "main" file - the file, which contains complete module registration

The main advantage of this mode is that you don't have to recompile the whole project if only single declaration was changed. Thus this mode suites well huge projects.

There are few problems with this mode:

1. There are use cases, when the generated file name is too long. *Py++* uses class name as a basis for the file name. So in case of template instantiations the file name could be really long, very long.
2. This mode doesn't play nicely with IDEs. Every time you add/remove classes in your project the list of generated files will be changed. So, you will have to maintain your IDE environment file.

   This problem was addressed in "fixed set of multiple files" mode. Keep reading :-).
3. If your project has pretty big class, than it is possible that the generated code will be too big and it take huge amount of time to compile it (GCC) or even to fail to compile it (MSVC 7.1).

   This problem was addressed in "multiple files, where single class code is split to few files" mode.

Usage example

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t(...)
mb.build_code_creator( ... )
mb.split_module( <<<directory name>>> )
```

Multiple files, where single class code is split to few files

This mode solves the problem, I mentioned earlier - you have to expose huge class and you have problems to compile generated code.

Py++ will split huge class to files using the following strategy:

- every generated source file can contain maximum 20 exposed declarations
- the following declarations are split to separate source files:
  - enumerations
  - unnamed enumerations
  - classes
  - member functions
  - virtual member functions
  - pure virtual member functions
  - protected member functions
- "main" class file - the file, which contains complete definition/registration of the generated file

Usage example

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t(...)
mb.build_code_creator( ... )
mb.split_module( <<<directory name>>>, [ <<<list of huge classes names>>> ] )
```

Fixed set of multiple files

This mode was born to play nicely with IDEs. It also can solve the problem with long file names. The scheme used to name files doesn't use class name.

In this mode you define the number of generated source files for classes.

Usage example

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t(...)
mb.build_code_creator( ... )
mb.balanced_split_module( <<<directory name>>>, <<<number of generated source files>>>
↪ )
```

Precompiled header

Usage of precompiled header file reduces overall compilation time. Not all compilers support the feature, moreover some of them can't handle presence of "boost/python.hpp" header in precompiled header file.

Py++ doesn't provide user-friendly API to add/define precompiled header file to the generated code. The main reason is that I don't have a good idea how to integrate/add this functionality to *Py++*. Nevertheless, you can enjoy from this time-saving feature:

```
from pyplusplus import module_builder
from pyplusplus import code_creators

mb = module_builder_t( ... )
mb.build_code_creator( ... )

precompiled_header = code_creators.include_t( 'your file name' )
mb.code_creator.adopt_creator( precompiled_header, 0 )

mb.split_module( ... )
```

API summary

Class `module_builder_t` contains 3 functions, related to file generation:

- ```
  def write_module( file_name )
  ```

- ```
  def split_module( self
                    , dir_name
                    , huge_classes=None
                    , on_unused_file_found=os.remove
                    , use_files_sum_repository=True)
  ```

  - `dir_name` - directory name the generated files will be put in
  - `huge_classes` - list of names of huge classes
  - `on_unused_file_found` - callable object, which is called every time *Py++* found that previously generated file is not in use anymore.
  - `use_files_sum_repository` *Py++* is able to store md5 sum of the generated files in a file. Next time you will generate code, Py++ will compare generated file content against the sum, instead of loading the content of the previously generated file from the disk and comparing against it.

    "<your module name>.md5.sum" is the file, that will be generated in the `dir_name` directory.

    Enabling this functionality should give you 10-15% of performance boost.

    Warning: If you changed manually some of the files - don't forget to delete the relevant line from "md5.sum" file. You can also delete the whole file. If the file is missing, Py++ will use old plain method of comparing content of the files. It will not re-write "unchanged" files and you will not be forced to recompile the whole project.

- ```
  def balanced_split_module( self
                             , dir_name
                             , number_of_files
                             , on_unused_file_found=os.remove
                             , use_files_sum_repository=True)
  ```

  - `number_of_files` - the desired number of generated source files
  ```

4.1.7 Py++ warnings

Introduction

Py++ has been created with few goals in mind:

- to allow users create Python bindings for large projects using the Boost.Python library
- to minimize maintenance time
- to serve as a user's guide for Boost.Python library

Those goals all have something in common. In order to achieve them, *Py++* must give useful feedback to the user. Because *Py++* understands the declarations it exports, it can scan declarations for potential problems, report them and in some cases provide hints about how to resolve the problem. Few examples:

- ```
  struct Y{ ... };
  ```

  ```
  struct X{
      ...
      virtual Y& do_smth();
  };
  ```

  Member function `do_smth` cannot be overridden in Python because ... .

- ```
  struct window{
      ...
      void get_size( int& height, int& width ) const;
  };
  ```

  Member function `get_size` can be exposed to Python, but it will not be callable because ... .

- In order to expose free/member function that takes more than 10 arguments user should define `BOOST_PYTHON_MAX_ARITY` macro.

- ```
  struct X{
      ...
  };

  void do_smth( X x );
  ```

  If you expose `do_smth` function and don't expose struct `X`, *Py++* will tell you that struct `X` is used in exported declaration, but was not exposed.

For these problems and many other *Py++* gives a nice explanation and sometimes a link to the relevant information on the Internet.

I don't know what about you, but I found these messages pretty useful. They allow me to deliver Python bindings with higher quality.

How it works?

In previous paragraph, I described some pretty useful functionality but what should you do to enable it? - *Nothing!* By default, *Py++* only prints the important messages to `stdout`. More over, it prints them only for to be exposed declarations.

Py++ uses the python logging package to write all user messages. By default, messages with `DEBUG` level will be skipped, all other messages will be reported.
  ```

Warnings

Example of the warning:

```
WARNING: containers::item_t [struct]
> warning W1020: Py++ will generate class wrapper - hand written code
> should be added to the wrapper class
```

Almost every warning reported by *Py++* consists from 3 parts:

- description of the declaration it refers to: "containers::item_t [struct]"
- warning unique identifier: "W1020"
- short explanation of the problem: "Py++ will generate class wrapper - hand written code should be added to the wrapper class"

API Description

How to disable warning(s)?

Every warning has unique identifier. In the example I gave it was `W1020`.

```
from pyplusplus import messages
from pyplusplus import module_builder

mb = module_builder.module_builder_t( ... )
xyz = mb.class_( XYZ )
xyz.disable_warnings( messages.W1020 )
```

It is also possible to disable warnings for all declarations. `pyplusplus.messages` package defines `DISABLE_MESSAGES` variable. This variable(`list`) keeps all warnings, which should not be reported. Use `messages.disable` function to edit it:

```
messages.disable( messages.W1020 )

#you also can disable warnings reporting at all:
messages.disable( *messages.all_warning_msgs )
```

Logging API

If you are here, it probably means that you are not pleased with default configuration and want to change it, right?

1. If you simply want to change the logging message level:

   ```
   import logging
   from pyplusplus import module_builder
   ```

   ```
   module_builder.set_logger_level( logging.DEBUG )
   ```

2. But what if you want to disable some messages and leave others? This is also possible. *Py++* and pygccxml do not use a single logger. Almost every internal package has its own logger. So you can enable one logger and disable another one.

   The pygccxml package defines all loggers in the `pygccxml.utils` package.

The *Py++* package defines all loggers in the `pyplusplus._logging_` package.

Both packages define a `loggers` class. Those classes keep references to different loggers. The `loggers` classes look very similar to the following class:

```
import logging #standard Python package

def _create_logger_( name ):
    logger = logging.getLogger(name)
    ...
    return logger

class loggers:
     file_writer = _create_logger_( 'pyplusplus.file_writer' )
     declarations = _create_logger_( 'pyplusplus.declarations' )
     module_builder = _create_logger_( 'pyplusplus.module_builder' )
     root = logging.getLogger( 'pyplusplus' )
     all = [ root, file_writer, module_builder, declarations ]
```

You can use these references in the `logging` package to complete your task of adjusting individual loggers.

One more thing, *Py++* automatically splits long message, where line length defaults to 70 characters. Thus it is very convenient to read them on your screen. If you want to use different tools to monitor those messages, consider to use standard Formatter class, instead of `multi_line_formatter_t` one.

Declarations API

Every declaration class has the following methods:

- `why_not_exportable( self )`

  This method explains why a declaration could not be exported. The return value is a string or `None`. `None` is returned if the declaration is exportable.

  Property `exportable` will be set to `True` if declaration is exportable, and to `False` otherwise.

- `readme( self )`

  This method gives you access to all tips/hints/warnings *Py++* has about the declaration. This methods returns a list of strings. If the declaration is not exportable, than first message within the list is an explanation, why it is not exportable.

4.1.8 C++ containers support

Introduction

C++ has a bunch of container classes:

- list
- deque
- queue
- priority_queue
- vector
- stack

- map
- multimap
- hash_map
- hash_multimap
- set
- hash_set
- multiset
- hash_multiset

It is not a trivial task to expose C++ container to Python. Boost.Python has a functionality that will help you to expose some of STL containers to Python. This functionality called - "indexing suite". If you want, you can read more about indexing suite here.

Boost.Python, out of the box, supports only `vector`, `map` and `hash_map` containers. In October 2003, Raoul Gough implemented support for the rest of containers. Well, actually he did much more - he implemented new framework. This framework provides support for almost all C++ containers and also an easy way to add support for custom ones. You'd better read his post to Boost.Python mailing list or *documentation* for the new indexing suite.

Now, I am sure you have the following question: if this suite is so good, why it is not in the main branch? The short answer is that this suite has some problems on MSVC 6.0 compiler and there are few users, that still use that compiler. The long answer is here:

- http://mail.python.org/pipermail/c++-sig/2006-June/010830.html
- http://mail.python.org/pipermail/c++-sig/2006-June/010835.html

Py++ and indexing suites

Py++ implements support for both indexing suites. More over, you can freely mix indexing suites. For example you can expose `std::vector<int>` using Boost.Python built-in indexing suite and `std::map< int, std::string>` using Raoul Gough's indexing suite.

How does it work?

In both cases, *Py++* provides almost "hands free" solution. *Py++* keeps track of all exported functions and variables, and if it sees that there is a usage of stl container, it exports the container. In both cases, *Py++* analyzes the container `value_type` (or in case of mapping container `mapped_type`), in order to set reasonable defaults, when it generates the code.

Indexing suite version 2 installation

None :-)

Py++ version 1.1, introduceds few breaking changes to this indexing suite:

- the suite implements all functionality in the header files only. Few *.cpp* files were dropped
- header files include directive was changed from

```
#include "boost/python/suite/indexing/..."
```

  to

```
#include "indexing_suite/..."
```

The change was done to simplify the indexing suite installation and redistribution. The gain list:

- no need to deal with patching and rebuilding Boost
- it is possible to use Boost libraries, which comes with your system
- you can put the library anywhere you want - just update the include paths in your build script
- it is easier to redistribute it - just include the library with your sources
- If you are a happy *Py++* user:
  - *Py++* will generate the indexing suite source files in the "generated code" directory, under *indexing_suite* directory.
  - *Py++* will take care to upgrade the files

The bottom line: *Py++* makes C++ STL containers handling fully transparent for its users.

Indexing suites API

By default, *Py++* works with built-in indexing suite. If you want to use indexing suite version 2, you should tell this to the `module_builder_t.__init__` method:

```
mb = module_builder_t( ..., indexing_suite_version=2 )
```

Every declared class has `indexing_suite` property. If the class is an instantiation of STL container, this property contains reference to an instance of `indexing_suite1_t` or `indexing_suite2_t` class.

How does *Py++* know, that a class represents STL container instantiation? Well, it uses `pygccxml.declarations.container_traits` to find out this. `pygccxml.declarations.container_traits` class, provides all functionality needed to identify container and to find out its `value_type` (`mapped_type`).

Built-in indexing suite API

Py++ defines `indexing_suite1_t` class. This class allows configure any detail of generated code:

- `no_proxy` - a boolean, if `value_type` is one of the the following types
  - fundamental type
  - enumeration
  - std::string or std::wstring
  - boost::shared_ptr<?>

  then, `no_proxy` will be set to `True`, otherwise to `False`.
- `derived_policies` - a string, that will be added as is to generated code
- `element_type` - is a reference to container `value_type` or `mapped_type`.

Indexing suite version 2 API

Please read "Indexing Suite V2" *introduction*, before proceeding with this section.

In this case there is no single place, where you can configure exported container functionality. Please take a look on the following C++ code:

```
struct item{
    ...
private:
    bool operator==( const item& ) const;
    bool operator<( const item& ) const;
};

struct my_data{
    std::vector<item> items;
    std::map< std::string, item > name2item_mapping;
};
```

Py++ declarations tree will contains `item`, `my_data`, `vector<item>` and `map<string,item>` class declarations.

If `value_type` does not support "equal" or "less than" functionality, sort and search functionality could not be exported.

Py++ class declaration has two properties: `equality_comparable` and `less_than_comparable`. The value of those properties is calculated on first invocation. If *Py++* can find `operator==`, that works on `value_type`, then, `equality_comparable` property value will be set to `True`, otherwise to `False`. Same process is applied on `less_than_comparable` property.

In our case, *Py++* will set both properties to `False`, thus sort and search functionality will not be exported.

It is the time to introduce `indexing_suite2_t` class:

- `container_class` - read only property, returns reference to container class declaration
- `container_traits` - read only property, returns reference to the relevant container traits class. Container traits classes are defined in `pygccxml.declarations` package.
- `element_type` - is a reference to container `value_type` or `mapped_type`.
- `call_policies` - read/write property, in near future I will add code to *Py++* that will analyze container `value_type` and will decide about default call policies. Just an example: for non-copy constructable classes `call_policies` should be set to `return_internal_reference`.
- `[disable|enable]_method` - new indexing suite, allows one to configure functionality exported to Python, using simple bitwise operations on predefined flags. *Py++* allows you to specify what methods you want to disable or enable. `indexing_suite2_t.METHODS` contains names of all supported methods.
- `[disable|enable]_methods_group` - almost same as above, but allows you to specify what group of methods you want to disable or enable. `indexing_suite2_t.METHOD_GROUPS` contains names of all supported groups.

Small tips/hints

1. If you set `equality_comparable` or `less_than_comparable` to `False`. The indexing suite will disable relevant functionality. You don't have explicitly to disable method or methods group.
2. The documentation of new indexing suite contains few small mistakes. I hope, I will have time to fix them. Any way, *Py++* generates correct code.

Indexing Suite V2

4.1.9 ctypes integration

Introduction

Boost.Python is really a very powerful library, but if you are working with code written in plain "C" - you've got a problem. You have to create wrappers for almost every function or variable.

In general, if you want to work with plain "C" code from Python you don't have to create any wrapper - you can use ctypes package.

About ctypes

ctypes is a foreign function library for Python. It provides C compatible data types, and allows one to call functions in dlls/shared libraries. It can be used to wrap these libraries in pure Python.

The idea

The idea behind "ctypes integration" functionality is really simple: you configure *Py++* to expose address of the variable\return value, and than you you use ctypes from_address functionality to access and modify the data.

Obviously, this approach has pros and cons:

- cons - it could be very dangerous - you can corrupt your application memory
- cons - managing memory is not something a typical Python user get used to. It is too "low level".
- pros - you don't need to create wrapper in C++
- pros - a Python user has access to the data
- pros - compilation time is smaller
- pros - you still can create wrapper, but using Python

In my opinion, the better way to go is to "mix":

1. expose your native code using Boost.Python and "ctypes integration" functionality - it is easy and cheap
2. use ctypes module to access your data
3. create high level API in Python: the wrappers, which will ensure the constraints and will provide more "natural" interface

Implemented functionality

Py++ is able to

- expose global and member variable address
- expose "this" pointer value
- expose a class "sizeof"
- expose variable, which has a union type
- return address of return value as integer - *new call policy was created*

ctypes integration contents

Variables

expose_address

`variable_t` declarations have got new property `expose_address`. If you set it value to `True`, *Py++* will register new property with the same name, but the type of it will be `unsigned int` and the value is address of the variable.

Py++ will take care and generate the right code for global, static and member variables.

Show me the code

Lets say you have the following C++ code:

```
struct bytes_t{
    bytes_t(){
        data = new int[5];
        for(int i=0; i<5; i++){
            data[i] = i;
        }
    }
   ...
   int* data;
   static int* x;
};

//somewhere in a cpp file
int* bytes_t::x = new int( 1997 );
```

In order to get access to the `bytes_t::data` and `bytes_t::x` you have to turn on `expose_address` property to `True`:

```
mb = module_builder_t( ... )
bytes = mb.class_( 'bytes_t' )
bytes.vars().expose_address = True
```

Py++ will generate code, which will expose the address of the variables.

and now it is a time to show some ctypes magic:

```
import ctypes
import your_module as m

bytes = m.bytes_t()

data_type = ctypes.POINTER( ctypes.c_int )
data = data_type.from_address( bytes.data )
for j in range(5):
    print '%d : %d' % ( j, data[j] )

data_type = ctypes.POINTER( ctypes.c_int )
data = data_type.from_address( m.bytes_t.x )
print x.contents.value
```

this & sizeof

The purpose

Py++ can expose a class `sizeof` and `this` pointer value to Python. I created this functionality without special purpose in mind.

Example

```
mb = module_builder_t( ... )
cls = mb.class_( <<<your class>>> )
cls.expose_this = True
cls.expose_sizeof = True
```

The Python class will contain two properties `this` and `sizeof`. The usage is pretty simple:

```
import ctypes
from <<<your module>>> import <<<your class>>> as data_t

d = data_t()
print d.this
print d.sizeof
```

Warning: I hope you know what you are doing, otherwise don't blame me :-)

C++ union

Introduction

Boost.Python does not help you to expose a variable, which has a union type. In this document, I am going to show you a complete example how to get access to the data, stored in the variable.

Py++ will not expose a union - it is impossible using Boost.Python, instead it will expose the address of the variable and the rest is done from the Python using ctypes package.

Example

For this example I am going to use the following code:

```
struct data_t{
    union actual_data_t{
        int i;
        double d;
    };
    actual_data_t data;
};
```

As in many other cases, *Py++* does the job automatically:

```
mb = module_builder_t( ... )
mb.class_( 'data_t' ).include()
```

no special code, to achieve the desired result, was written.

The generated code is boring, so I will skip it and will continue to the usage example:

```
import ctypes
from <<<your module>>> import data_t

#lets define our union
class actual_data_t( ctypes.Union ):
    _fields_ = [( "i", ctypes.c_int ), ( 'd', ctypes.c_double )]

obj = data_t()
actual_data = actual_data_t.from_address( obj.data )
#you can set\get data
actual_data.i = 18
prit actual_data.i
actual_data.d = 12.12
print actual_data.d
```

That's all. Everything should work fine. You can add few getters and setters to class `data_t`, so you could verify the results. I did this for a tester, that checks this functionality.

Future directions

The functionality is going to be developed father and I intend to add the following features:

- to port this functionality to 64bit systems
- to add ability to expose "C" functions without using Boost.Python.

4.1.10 Functions & operators

Preamble

Boost.Python provides very rich interface to expose functions and operators. This section of documentation will explain how to configure *Py++* in order to export your functions, using desired Boost.Python functionality.

Contents

Call policies

Introduction

Boost.Python has a nice introduction to call policies. "Call policies concept" document will provide you with formal definition.

Syntax

The call policies in *Py++* are named exactly as in Boost.Python, only the syntax is slightly different. For instance, this call policy:

```
return_internal_reference< 1, with_custodian_and_ward<1, 2> >()
```

becomes in *Py++*

```
return_internal_reference( 1, with_custodian_and_ward(1, 2) )
```

Py++ supports all call policies presented in Boost.Python.

Usage example

Every "callable" object in *Py++* has `call_policies` property.

C++ code:

```
struct data{...};
const data& do_smth( const data& d, int x );

void return_second_arg( int x, int y );

typedef struct opaque_ *opaque_pointer;
opaque_pointer get_opaque();
```

Python code:

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

mb = module_builder.module_builder_t( ... )
mb.free_function( 'return_second_arg' ).call_policies = call_policies.return_
↪arg( 2 )
#--------------------------------------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
↪^^^^^^^^

mb.member_function( 'do_smth' ).call_policies = call_policies.return_self()
#-------------------------------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

mb.calldef( 'get_opaque' ).call_policies
    = call_policies.return_value_policy( call_policies.return_opaque_pointer␣
↪)
```

Defaults

Py++ is able to "guess" few call policies, base on analysis of return type and\or callable name:

- `default_call_policies`:
  - Python immutable type returned by value: C++ fundamental types, `std::string`, enumerations
  - user-defined type (class) returned by value
  - return type is `const char*`
- `return_value_policy`
  - `return_opaque_pointer`
    - return type is `void*`
    - return type is `const void*`

        * return type is `T*` and `T` is a user defined opaque type

          `class_t` and `class_declaration_t` classes have `opaque` property. You have to set it to `True`, if you want *Py++* to create this call policy automatically for all functions, that use `T*` as return type.

    - `copy_const_reference`
        * return type is `const T&`
        * for member `operator[]` that returns const reference to immutable type
    - `return_by_value`
        * return type is `const wchar_t*`
    - `copy_non_const_reference`
        * return type is `T&`, for member `operator[]` that returns reference to immutable type
- `return_internal_reference`
    - return type is `T&`, for member `operator[]`
- `return_self`

  This call policy will be used for `operator=`.

Missing call policies

If you don't specify call policy for a function and it needs one, few things will happen:

- *Py++* prints a warning message
- *Py++* generates code with

```
/* undefined call policies */
```

  comment, instead of call policy. If *Py++* was wrong and function doesn't need call policy the generate code will compile fine, otherwise you will get a compilation error.

Special case

Before you read this paragraph consider to read Boost.Python return_opaque_pointer documentation.

`return_value_policy( return_opaque_pointer )` is a special policy for Boost.Python. In this case, it requires from you to define specialization for the `boost::python::type_id` function on the type pointed to by returned pointer. *Py++* will generate the required code.

Actually you should define `boost::python::type_id` specialization also in case a function takes the opaque type as an argument. *Py++* can do it for you, all you need is to mark a declaration as opaque.

Example:

```
struct identity_impl_t{};
typedef identity_impl_t* identity;

struct world_t{

    world_t( identity id );
```

```
    identity get_id() const;

    ...
};
```

Py++ code:

```
mb = module_builder_t(...)
mb.class_( 'identity_impl_t' ).opaque = True
```

Py++ defined call policies

Py++ defines few call policies. I hope you will find them useful. I don't mind to contribute them to Boost.Python library, but I don't have enough free time to "boostify" them.

as_tuple

Definition

Class `as_tuple` is a model of ResultConverterGenerator which can be used to wrap C++ functions returning a pointer to arrays with fixed size. The policy will construct a Python tuple from the array and handle the array memory.

Example

```
 struct vector3{
     ...

     float* clone_raw_data() const{
         float* values = new float[3];
         //copy values
         return values;
     }

     const flow* get_raw_data() const{
         return m_values;
     }

 private:
     float m_values[3];
 };

namespace bpl = boost::python;
namespace pypp_cp = pyplusplus::call_policies;
BOOST_PYTHON_MODULE(my_module){
  bpl::class_< vector3 >( "vector3" )
      .def( "clone_raw_data"
            , &::vector3::clone_raw_data
            , bpl::return_value_policy< pypp_cp::arrays::as_tuple< 3, pypp_cp::memory_
↪managers::delete_ > >() )
      .def( "get_raw_data"
            , &::vector3::get_raw_data
```

```
            , bpl::return_value_policy< pypp_cp::arrays::as_tuple< 3, pypp_cp::memory_
↪managers::none > >() ) );
}
```

as_tuple class

as_tuple is a template class that takes few arguments:

1. the array size - compile time constant
2. memory management policy - a class, which will manage the return value. There are two built-in memory managers:
   - delete_ - the array will be deleted after it was copied to tuple, using operator delete[]
   - none - do nothing

The *Py++* code is slightly different from the C++ one, but it is definitely shorter:

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

mb = module_builder.module_builder_t( ... )
mb.member_function( 'clone_raw_data' ).call_policies \
    = call_policies.convert_array_to_tuple( 3, call_policies.memory_managers.delete_ )
mb.member_function( 'get_raw_data' ).call_policies \
    = call_policies.convert_array_to_tuple( 3, call_policies.memory_managers.none )
```

return_addressof

Definition

Class return_addressof is a model of ResultConverterGenerator which can be used to wrap C++ functions returning any pointer, such that the pointer value is converted to unsigned int and it is copied into a new Python object.

This call policy was created to be used with ctypes package and provide access to some raw\low level data, without creating wrappers.

Pay attention: you have to manage the memory by your own.

Example

```
int* get_value(){
    static int buffer[] = { 0,1,2,3,4 };
    return buffer;
}

namespace bpl = boost::python;
BOOST_PYTHON_MODULE(my_module){
  def( "get_value"
       , bpl::return_value_policy< pyplusplus::call_policies::return_addressof<> >()␣
↪);
}
```

The *Py++* code is not that different from what you already know:

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

mb = module_builder.module_builder_t( ... )
mb.free_function( return_type='float *' ).call_policies \
    = call_policies.return_value_policy( call_policies.return_addressof )
```

Python code:

```
import ctypes
import my_module

buffer_type = ctypes.c_int * 5
buffer = buffer_type.from_address( my_module.get_value() )
assert [0,1,2,3,4] == list( buffer )
```

return_pointee_value

Definition

Class `return_pointee_value` is a model of ResultConverterGenerator which can be used to wrap C++ functions, that return a pointer to a C++ object. The policy implements the following logic:

```
if( <<<return value is NULL pointer>>> ){
    return None;
}
else{
    return boost::python::object( *<<<return value>>> );
}
```

The return type of the function should be `T*`.

It passes the value of the pointee to Python, thus the conversion for `T` is used. This call policy could be used to return pointers to Python, which types are not known to Boost.Python, but only a conversion for the pointees.

Therefore this policy should be used to return pointers to objects, whose types were wrapped with other tools, such as SWIGSIP.

Another usage of this call policy is to return to Python new object, which contains copy of `(*return value)`.

Please note: This policy does not take ownership of the wrapped pointer. If the object pointed to is deleted in C++, the python-object will become invalid too, if your custom conversion depends on the original object.

Examples

Unknown type

This technique and example was contributed by Maximilian Matthe.

```
struct int_wrapper{
     int_wrapper(int v)
     : val(v)
     {}
```

```
    int val;
};

//we will expose the following function
int_wrapper* return_int_wrapper(){
    static int_wrapper w(42);
    return &w;
}

//the Boost.Python custom converter
struct convert_int_wrapper{
    static PyObject* convert(int_wrapper const& w){
        boost::python::object value(w.val);
        return boost::python::incref( value.ptr() );
    }
};


BOOST_PYTHON_MODULE(my_module){
    using namespace boost::python;
    //register our custom converter
    to_python_converter<int_wrapper, convert_int_wrapper, false>();

    def( "return_int_wrapper"
         , &return_int_wrapper
         , return_value_policy<return_pointee_value>() );
}
```

Python code:

```
import my_module

assert 42 == my_module.return_int_wrapper()
```

Return pointee value

```
float* get_value(){
    static float value = 0.5;
    return &value;
}

float* get_null_value(){
  return (float*)( 0 );
}

namespace bpl = boost::python;
BOOST_PYTHON_MODULE(my_module){
  def( "get_value"
       , bpl::return_value_policy< pyplusplus::call_policies::return_pointee_value<> >
↪() );

  def( "get_null_value"
       , bpl::return_value_policy< pyplusplus::call_policies::return_pointee_value<> >
↪() );
}
```

The *Py++* code is not that different from what you already know:

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

mb = module_builder.module_builder_t( ... )
mb.free_function( return_type='float *' ).call_policies \
    = call_policies.return_value_policy( call_policies.return_pointee_value )
```

Python code:

```
import my_module

assert 0.5 == my_module.get_value()
assert None is my_module.get_null_value()
```

return_range

Definition

Class `return_range` is a model of CallPolicies, which can be used to wrap C++ functions that return a pointer to some array. The new call policy constructs object, which provides a regular Python sequence interface.

Example

```
struct image_t{

    ...

    const unsigned char* get_data() const{
        return m_raw_data;
    }

    ssize_t get_width() const{
        return m_width;
    }

    ssize_t get_height() const{
        return m_height;
    }

private:
    unsigned long m_width;
    unsigned long m_height;
    unsigned char* m_raw_data;
};
```

Before introducing the whole solution, I would like to describe “return_range” interface.

`return_range` class

```
template < typename TGetSize
           , typename TValueType
           , typename TValuePolicies=boost::python::default_call_policies >
struct return_range : boost::python::default_call_policies
{ ... };
```

Boost.Python call policies are stateless classes, which do not care any information about the invoked function or object. In our case we have to pass the following information:

- the array size
- the array type
- "__getitem__" call policies for the array elements

`TGetSize` parameter

`TGetSize` is a class, which is responsible to find out the size of the returned array.

`TGetSize` class must have:

- default constructor
- call operator with the following signature:

```
ssize_t operator()( boost::python::tuple args );
```

`args` is a tuple of arguments, the function was called with.

Pay attention: this operator will be invoked **after** the function. This call policy is **not thread-safe**!

For our case, the following class could be defined:

```
struct image_data_size_t{
    ssize_t operator()( boost::python::tuple args ){
        namespace bpl = boost::python;
        bpl::object self = args[0];
        image_t& img = bpl::extract< image_t& >( self );
        return img.get_width() * img.get_height();
    }
};
```

Passing all arguments, instead of single "self" argument gives you an ability to treat functions, where the user asked to get access to the part of the array.

```
struct image_t{
    ...
    const unsigned char* get_data(ssize_t offset) const{
        //check that offset represents a legal value
        ...
        return &m_raw_data[offset];
    }
    ...
};
```

The following "get size" class treats this situation:

```
struct image_data_size_t{
    ssize_t operator()( boost::python::tuple args ){
        namespace bpl = boost::python;
        bpl::object self = args[0];
        image_t& img = bpl::extract< image_t& >( self );
        bpl::object offset_obj = args[1];
        ssize_t offset = bpl::extract< ssize_t >( offset_obj );
        return img.get_width() * img.get_height() - offset;
    }
};
```

`TValueType` parameter

`TValueType` is a type of array element. In our case it is `unsigned char`.

`TValuePolicies` parameter

`TValuePolicies` is a "call policy" class, which will be applied when the array element is returned to Python. This is a call policy for "__getitem__" function.

`unsigned char` is mapped to immutable type in Python, so I have to use `default_call_policies`. `default_call_policies` is a default value for `TValuePolicies` parameter.

I think, now you are ready to see the whole solution:

```
namespace bpl = boost::python;
namespace ppc = pyplusplus::call_policies;

BOOST_PYTHON_MODULE(my_module){
  bpl::class_< image_t >( "image_t" )
      .def( "get_width", &image_t::get_width )
      .def( "get_height", &image_t::get_height )
      .def( "get_raw_data", ppc::return_range< image_size_t, unsigned char >() );
}
```

Py++ integration

The *Py++* code is not that different from what you already know:

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

image_size_code = \
"""
struct image_data_size_t{
    ssize_t operator()( boost::python::tuple args ){
        namespace bpl = boost::python;
        bpl::object self = args[0];
        image_t& img = bpl::extract< image_t& >( self );
        return img.get_width() * img.get_height();
    }
};
"""
```

```
mb = module_builder.module_builder_t( ... )
image = mb.class_( 'image_t' )
image.add_declaration_code( image_size_code )
get_raw_data = image.mem_fun( 'get_raw_data' )
get_raw_data.call_policies \
    = call_policies.return_range( get_raw_data, "image_data_size_t" )
```

call_policies.return_range arguments:

1. A reference to function. *Py++* will extract by itself the type of the array element.
2. A name of "get size" class.
3. A call policies for "__getitem__" function. *Py++* will analyze the array element type. If the type is mapped to immutable type, than `default_call_policies` is used, otherwise you have to specify call policies.

Python usage code:

```
from my_module import *

img = image_t(...)
for p in img.get_raw_data():
    print p
```

Dependencies

The new call policy depends on *new indexing suite* and *Py++* :-).

custom_call_policies

Definition

`custom_call_policies` is a special call policy, which allows you to integrate the call policies, you defined, with *Py++*

Example

```
from pyplusplus import module_builder
from pyplusplus.module_builder import call_policies

mb = module_builder.module_builder_t( ... )
mb.free_function( ... ).call_policies \
    = call_policies.custom_call_policies( <<<your call policies code>>> )

Optionally you can specify name of the header file, which should be included:
```

```
mb.free_function( ... ).call_policies \
   = call_policies.custom_call_policies( <<<your call policies code>>>, "xyz.hpp" )
```

Function transformation

Introduction

During the development of Python bindings for some C++ library, it might get necessary to write custom wrapper code for a particular function in order to make that function usable from Python.

An often mentioned example that demonstrates the problem is the `get_size()` member function of a fictitious image class:

```
void get_size(int& width, int& height);
```

This member function cannot be exposed with standard Boost.Python mechanisms. The main reasons for this is that `int` is immutable type in Python. An instance of immutable type could not be changed after construction. The only way to expose this function to Python is to create small wrapper, which will return a tuple. In Python, the above function would instead be invoked like this:

```
width, height = img.get_size()
```

and the wrapper could look like this:

```
boost::python::tuple get_size( const image_t& img ){
    int width;
    int height;
    img.get_size( width, height );
    return boost::python::make_tuple( width, height );
}
```

As you can see this function is simply invokes the original `get_size()` member function and return the output values as a tuple.

Unfortunately, C++ source code cannot describe the semantics of an argument so there is no way for a code generator tool such as *Py++* to know whether an argument that has a reference type is actually an output argument, an input argument or an input/output argument. That's why the user will always have to "enhance" the C++ code and tell the code generator tool about the missing information.

Note: C++ fundamental types, enumerations and string are all mapped to Python immutable types.

Instead of forcing you to write the entire wrapper function, *Py++* allows you to provide the semantics of an argument(s) and then it will take care of producing the correct code:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
get_size = mb.mem_fun( 'image_t::get_size' )
get_size.add_transformation( FT.output(0), FT.output(1) )
#the following line has same effect
get_size.add_transformation( FT.output('width'), FT.output('height') )
```

Py++ will generate a code, very similar to one found in `boost::python::tuple get_size( const image_t& img )` function.

Thanks to

A thanks goes to Matthias Baas for his efforts and hard work. He did a research, implemented the initial working version and wrote a lot of documentation.

Transformers contents

Py++ comes with few predefined transformers:

Terminology

Function transformation

Py++ sub-system\framework, which allows you to create function wrappers and to keep smile.

The operation of changing one function into another in accordance with some rules. Especially: a change of return type and\or arguments and their mapping to the original ones.

Function wrapper (or just wrapper)

C++ function, which calls some other function.

Immutable type

An instance of this type could not be modified after construction

Transformer

An object that applies predefined set of rules on a function, during function-wrapper construction process.

Function alias (or just alias)

Name under which Python users see the exposed function

Name mangling

Definition

Wikipedia has a nice explanation what name mangling is.

Why?

I am sure you want to ask why and where *Py++* uses name mangling? *Py++* uses name mangling to create function-wrappers for overloaded and\or free functions. Consider the following use case:

```
void get_distance( long& );
void get_distance( double& );
```

In order to expose `get_distance` functions you have to create 2 function wrappers:

```
long get_distance_as_long(){...}
double get_distance_as_double(){...}
```

You have to give them distinguish names - C++ does not allow overloading, base on return type only. You also have to exposes them under different aliases, otherwise they will not be callable from Python:

```
namespace bp = boost::python;
BOOST_PYTHON_MODULE(...){
    bp::def( "get_distance_as_long", &get_distance_as_long );
    bp::def( "get_distance_as_double", &get_distance_as_double );
}
```

The solution

Py++ implements a solution to the problem. The generated wrapper names are unique in the whole project. However, they are pretty ugly:

- `get_distance_610ef0e8a293a62001a25cd3dc59b769` for `get_distance( long& )` function
- `get_distance_702c7b971ac4e91b12f260ac85b36d84` for `get_distance( double& )` function

The good news - they will not be changed between different runs of the code generator.

If you are exposing an overloaded function, in that case *Py++* uses the ugly function-wrapper name as an alias. It is up to you to change the alias:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
get_distance_as_long = mb.mem_fun( 'get_distance', arg_types=['long &'] )
get_distance_as_long.add_transformation( FT.output(0), alias="get_distance_
↪as_long" )
```

There are two main reasons for such implementation\behaviour:

1. The generated code will always compile and be correct.
2. If you forgot to give an alias to a function, your users will still be able to call the function. So no need to rush and create new release.

`output` transformer

Definition

"output" transformer removes an argument from the function definition and adds the "returned", by the original function, value to the return statement of the function-wrapper.

"output" transformer takes as argument name or index of the original function argument. The argument should have "reference" type. Support for "pointer" type will be added pretty soon.

Example

```
#include <string>

inline void hello_world( std::string& hw ){
    hw = "hello world!";
}
```

Lets say that you need to expose `hello_world` function. As you know `std::string` is mapped to Python string, which is immutable type, so you have to create small wrapper for the function. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT
```

```
mb = module_builder.module_builder_t( ... )
hw = mb.mem_fun( 'hello_world' )
hw.add_transformation( FT.output(0) )
```

What you see below is the relevant pieces of generated code:

```
namespace bp = boost::python;

static boost::python::object hello_world_a3478182294a057b61508c30b1361318(  )
↪{
    std::string hw2;
    ::hello_world(hw2);
    return bp::object( hw2 );
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::def( "hello_world", &hello_world_a3478182294a057b61508c30b1361318 );
 }
```

`input` transformer

Definition

"input" transformer removes a "reference" type from the function argument.

"input" transformer takes as argument name or index of the original function argument. The argument should have "reference" type. Support for "pointer" type will be added pretty soon.

Example

```
#include <string>

inline void hello_world( std::string& hw ){
    hw = "hello world!";
}
```

Lets say that you need to expose `hello_world` function. As you know `std::string` is mapped to Python string, which is immutable type, so you have to create small wrapper for the function. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
hw = mb.mem_fun( 'hello_world' )
hw.add_transformation( FT.input(0) )
```

What you see below is the relevant pieces of generated code:

```
namespace bp = boost::python;

static void hello_world_a3478182294a057b61508c30b1361318( ::std::string hw ){
    ::hello_world(hw);
```

```
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::def( "hello_world", &hello_world_a3478182294a057b61508c30b1361318 );
}
```

`inout` transformer

Definition

`inout` transformer is a combination of *input* and *output* transformers. It removes a "reference" type from the function argument and then adds the "returned", by the original function, value to the return statement of the function-wrapper.

`inout` transformer takes as argument name or index of the original function argument. The argument should have "reference" type. Support for "pointer" type will be added pretty soon.

Example

```
#include <string>

inline void hello_world( std::string& hw ){
    hw = "hello world!";
}
```

Lets say that you need to expose `hello_world` function. As you know `std::string` is mapped to Python string, which is immutable type, so you have to create small wrapper for the function. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
hw = mb.mem_fun( 'hello_world' )
hw.add_transformation( FT.inout(0) )
```

What you see below is the relevant pieces of generated code:

```
namespace bp = boost::python;

static boost::python::object hello_world_a3478182294a057b61508c30b1361318(␣
↪::std::string hw ){
    ::hello_world(hw);
    return bp::object( hw );
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::def( "hello_world", &hello_world_a3478182294a057b61508c30b1361318 );
}
```

`modify_type` transformer

Definition

"modify_type" transformer changes type of the function argument.

"modify_type" transformer takes two arguments:

1. name or index of the original function argument
2. a callable, which takes as argument reference to type and returns new type

New in version greater than 0.8.5.

Pay attention!

If implicit conversion between new type and the old one does not exist "reinterpret_cast" will be used.

Example

```
#include <string>

inline void hello_world( std::string& hw ){
    hw = "hello world!";
}
```

Lets say that you need to expose `hello_world` function. As you know `std::string` is mapped to Python string, which is immutable type, so you have to create small wrapper for the function. The following *Py++* code does it for you:

```
from pygccxml import declarations
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
hw = mb.mem_fun( 'hello_world' )
hw.add_transformation( FT.modify_type(0, declarations.remove_reference) )
```

What you see below is the relevant pieces of generated code:

```
namespace bp = boost::python;

static void hello_world_a3478182294a057b61508c30b1361318( ::std::string hw ){
    ::hello_world(hw);
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::def( "hello_world", &hello_world_a3478182294a057b61508c30b1361318 );
}
```

`input_static_array` transformer

Definition

"input_static_array" transformer works on native static arrays. It handles the translation between Python object, passed as argument that represent a sequence, and the array. Size of array should be predefined.

"input_static_array" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second argument should be an integer value, which represents array size.

Example

```
struct vector3{

    void init( int values[3] ){
        x = values[0];
        y = values[1];
        z = values[2];
    }

    int x,y,z;
};
```

In order to expose `init` member function we need to create small wrapper: The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
v3 = mb.class_( 'vector3' )
v3.mem_fun( 'init' ).add_transformation( FT.input_static_array( 0, 3 ) )
```

What you see below is the relevant pieces of generated code:

```
#include "__convenience.pypp.hpp" //Py++ header file, which contains few␣
↪convenience function

namespace bp = boost::python;

static void init_418e52f4a347efa6b7e123b96f32a73c( ::ft::vector3 & inst,␣
↪boost::python::object values ){
    int native_values[3];
    pyplus_conv::ensure_uniform_sequence< int >( values, 3 );
    pyplus_conv::copy_sequence( values, pyplus_conv::array_inserter( native_
↪values, 3 ) );
    inst.init(native_values);
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::class_< ft::vector3 >( "vector3", "documentation" )
   .def( "init"
         , &init_418e52f4a347efa6b7e123b96f32a73c
         , ( bp::arg("inst"), bp::arg("values") ) )
   .def_readwrite( "x", &ft::vector3::x )
   .def_readwrite( "y", &ft::vector3::y )
```

```
    .def_readwrite( "z", &ft::vector3::z );
}
```

output_static_array transformer

Definition

"output_static_array" transformer works on native static arrays. It handles the translation between array and Python list object. Size of array should be predefined.

"output_static_array" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second argument should an integer value, which represents array size.

Example

```
struct vector3{

    void get_values( int values[3] ){
        values[0] = x;
        values[1] = y;
        values[2] = z;
    }

    int x,y,z;
};
```

In order to expose `get_values` member function we need to create small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
v3 = mb.class_( 'vector3' )
v3.mem_fun( 'get_values' ).add_transformation( FT.output_static_array( 0, 3␣
↪) )
```

What you see below is the relevant pieces of generated code:

```
#include "__convenience.pypp.hpp" //Py++ header file, which contains few␣
↪convenience function

namespace bp = boost::python;

static boost::python::object get_values_22786c66e5973b70f714e7662e2aecd2(␣
↪::ft::vector3 & inst ){
   int native_values[3];
   boost::python::list py_values;
   inst.get_values(native_values);
   pyplus_conv::copy_container( native_values, native_values + 3, pyplus_
↪conv::list_inserter( py_values ) );
   return bp::object( py_values );
}
```

```
BOOST_PYTHON_MODULE(...){
    ...
    bp::class_< ft::vector3 >( "vector3", "documentation" )
       .def( "get_values"
             , &get_values_22786c66e5973b70f714e7662e2aecd2
             , ( bp::arg("inst") ) )
       .def_readwrite( "x", &ft::vector3::x )
       .def_readwrite( "y", &ft::vector3::y )
       .def_readwrite( "z", &ft::vector3::z );
}
```

`inout_static_array` transformer

Definition

`inout_static_array` transformer is a combination of *input* and *output* transformers. It allows one to call a C++ function, which takes an array using Python `list` class

"inout_static_array" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second argument specifies the array size.

Example

```
int sum_and_fill( int v[3], int value ){
    int result = v[0] + v[1] + v[2];
    v[0] = value;
    v[1] = value;
    v[2] = value;
    return result;
}
```

In order to expose `sum_and_fill` function we need to create a small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
sum_and_fill = mb.free_fun( 'sum_and_fill' )
sum_and_fill.add_transformation( ft.inout_static_array('v', 3) )
```

What you see below is the relevant pieces of generated code:

```
static boost::python::tuple sum_and_fill_2dd285a3344dbf7d71ffb7c78dd614c5(␣
↪boost::python::object v, int value ){
    int native_v[3];
    boost::python::list py_v;
    pyplus_conv::ensure_uniform_sequence< int >( v, 3 );
    pyplus_conv::copy_sequence( v, pyplus_conv::array_inserter( native_v, 3␣
↪) );
    int result = ::sum_and_fill(native_v, value);
    pyplus_conv::copy_container( native_v, native_v + 3, pyplus_conv::list_
↪inserter( py_v ) );
```

```
    return bp::make_tuple( result, py_v );
}

BOOST_PYTHON_MODULE(ft_inout_static_array){
    { //::ft::sum_and_fill

        typedef boost::python::tuple ( *sum_and_fill_function_type )(␣
↪boost::python::object,int );

        bp::def(
                  "sum_and_fill"
                  , sum_and_fill_function_type( &sum_and_fill_
↪2dd285a3344dbf7d71ffb7c78dd614c5 )
                  , ( bp::arg("v"), bp::arg("value") ) );
    }
}
```

`transfer_ownership` transformer

Definition

"transfer_ownership" transformer changes type of the function argument, from `T*` to `std::auto_ptr<T>`. This transformer was born to provide the answer to How can I wrap a function which needs to take ownership of a raw pointer? FAQ.

"transfer_ownership" transformer takes one argument, name or index of the original function argument. The argument type should be "pointer".

New in version greater than 0.8.5.

Example

```
struct resource_t{...};

void do_smth(resource_t* r){
    ...
}
```

Lets say that you need to expose "do_smth" function. According to the FAQ, you have to create small wrapper, which will take `std::auto_ptr` as an argument. The following *Py++* code does it for you:

```
from pygccxml import declarations
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )

resource = mb.class_( 'resource_t' )
resource.held_type = 'std::auto_ptr< %s >' % resource.decl_string
do_smth = mb.free_fun( 'do_smth' )
do_smth.add_transformation( FT.transfer_ownership( 0 ) )
```

What you see below is the relevant pieces of generated code:

```
namespace bp = boost::python;

static void do_smth_4cf7cde5fca92efcdb8519f8c1a4bccd( std::auto_ptr<␣
↪::resource_t > r ){
 ::do_smth(r.release());
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::def( "do_smth", &do_smth_4cf7cde5fca92efcdb8519f8c1a4bccd, ( bp::arg(
↪"r") ) );
}
```

`input_c_buffer` transformer

Definition

"input_c_buffer" transformer works on C buffers. It handles the translation between a Python sequence object and the buffer.

"input_c_buffer" transformer takes as first argument name or index of the "buffer" argument. The argument should have "array" or "pointer" type. The second argument should be name or index of another original function argument, which represents array size.

Example

```
struct file_t{
    void write( char* buffer, int size ) const;
};
```

In order to expose `write` member function we need to create small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
f = mb.class_( 'file_t' )
f.mem_fun( 'write' ).add_transformation( FT.input_c_buffer( 'buffer', 'size'␣
↪) )
```

What you see below is the relevant pieces of generated code:

```
#include "__convenience.pypp.hpp" //Py++ header file, which contains few␣
↪convenience function

#include <vector>

#include <iterator>

namespace bp = boost::python;

static void write_8883fea8925bad9911e6c5a4015ed106( ::file_t const & inst,␣
↪boost::python::object buffer ){
```

```
    int size2 = boost::python::len(buffer);
    std::vector< char > native_buffer;
    native_buffer.reserve( size2 );
    pyplus_conv::ensure_uniform_sequence< char >( buffer );
    pyplus_conv::copy_sequence( buffer, std::back_inserter( native_buffer),␣
↪boost::type< char >() );
    inst.write(&native_buffer[0], size2);
}

BOOST_PYTHON_MODULE(...){
    ...
    bp::class_< file_t >( "file_t" )
        .def(
              "write"
            , (void (*)( ::file_t const &,boost::python::object ))( &write_
↪8883fea8925bad9911e6c5a4015ed106 )
            , ( bp::arg("inst"), bp::arg("buffer") ) );
}
```

`from_address` transformer

Definition

"from_address" transformer allows integration with `ctypes` package. Basically it handles the translation between `size_t` value, which represents a pointer to some data and the exposed code. Thus you can use `ctypes` package to create the data and than pass it to the Boost.Python exposed function.

"from_address" transformer takes as first argument name or index of the "data" argument. The argument should have "reference" or "pointer" type.

Example

```
unsigned long
sum_matrix( unsigned int* matrix, unsigned int rows, unsigned int columns ){
    if( !matrix ){
        throw std::runtime_error( "matrix is null" );
    }
    unsigned long result = 0;
    for( unsigned int r = 0; r < rows; ++r ){
        for( unsigned int c = 0; c < columns; ++c ){
            result += *matrix;
            ++matrix;
        }
    }
    return result;
}
```

In order to expose `sum_matrix` function we need to create a small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT
```

```
mb = module_builder.module_builder_t( ... )
mb.free_function( 'sum_matrix' ).add_transformation( FT.from_address( 0 ) )
```

What you see below is the relevant pieces of generated code:

```
static boost::python::object sum_matrix_515b62fca9176ae4fffaf5fb118855dc(␣
↪unsigned int matrix, unsigned int rows, unsigned int columns ){
    long unsigned int result = ::sum_matrix(reinterpret_cast< unsigned int *␣
↪>( matrix ), rows, columns);
    return bp::object( result );
}

BOOST_PYTHON_MODULE(...){
    { //::sum_matrix

        typedef boost::python::object ( *sum_matrix_function_type )(␣
↪unsigned int,unsigned int,unsigned int );

        bp::def(
            "sum_matrix"
            , sum_matrix_function_type( &sum_matrix_
↪515b62fca9176ae4fffaf5fb118855dc )
            , ( bp::arg("matrix"), bp::arg("rows"), bp::arg("columns") )
            , "documentation" );
    }
}
```

And now the Python usage example:

```
import ctypes
import mymodule

rows = 10
columns = 7
matrix_type = ctypes.c_uint * columns * rows
sum = 0
counter = 0
matrix = matrix_type()
for r in range( rows ):
   for c in range( columns ):
        matrix[r][c] = counter
        sum += counter
        counter += 1
result = module.sum_matrix( ctypes.addressof( matrix ), rows, columns )
```

`input_static_matrix` transformer

Definition

"input_static_matrix" transformer works on native static 2D arrays. It handles the translation between Python object, passed as argument that represent a sequence of sequences, and the matrix. The number of rows and columns should be known in advance.

"input_static_matrix" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second and the third arguments specify rows and columns size.

Limitations

This transformer could not be applied on virtual functions.

Example

```
template< int rows, int columns >
int sum_impl( const int m[rows][columns] ){
    int result = 0;
    for( int r = 0; r < rows; ++r ){
        for( int c = 0; c < columns; ++c ){
            result += m[r][c];
        }
    }
    return result;
}

int sum( int m[2][3]){
    return sum_impl<2, 3>( m );
}
```

In order to expose `sum` function we need to create a small wrapper: The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
sum = mb.free_fun( 'sum' )
sum.add_transformation( FT.input_static_matrix('m', rows=2, columns=3) )
```

What you see below is the relevant pieces of generated code:

```
#include "__convenience.pypp.hpp" //Py++ header file, which contains few␣
↪convenience function

namespace bp = boost::python;

static boost::python::object sum_d4475c1b6a0ff117f0754ec5ecacdda3(␣
↪boost::python::object m ){
    int native_m[2][3];
    pyplus_conv::ensure_uniform_sequence< boost::python::list >( m, 2 );
    for( size_t row = 0; row < 2; ++row ){
        pyplus_conv::ensure_uniform_sequence< int >( m[row], 3 );
        pyplus_conv::copy_sequence( m[row], pyplus_conv::array_inserter(␣
↪native_m[row], 3 ) );
    }
    int result = ::ft::sum(native_m);
    return bp::object( result );
}

BOOST_PYTHON_MODULE(...){
    ...
   typedef boost::python::object ( *sum_function_type )(␣
↪boost::python::object );

   bp::def(
```

```
        "sum"
        , sum_function_type( &sum_d4475c1b6a0ff117f0754ec5ecacdda3 )
        , ( bp::arg("m") ) );
}
```

`output_static_matrix` transformer

Definition

"output_static_matrix" transformer works on native 2D static arrays. It handles the translation between a matrix and Python list object. The matrix row and column sizes should be known in advance.

"output_static_matrix" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second and the third arguments specify rows and columns size.

Limitations

This transformer could not be applied on virtual functions.

Example

```
void filler( int m[2][3], int value ){
    for( int r = 0; r < 2; ++r ){
        for( int c = 0; c < 3; ++c ){
            m[r][c] = value;
        }
    }
}
```

In order to expose `filler` function we need to create a small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
filler = mb.free_fun( 'filler' )
filler.add_transformation( ft.output_static_matrix('m', rows=2, columns=3) )
```

What you see below is the relevant pieces of generated code:

```
#include "__convenience.pypp.hpp" //Py++ header file, which contains few␣
↪convenience function

namespace bp = boost::python;

static boost::python::object filler_7b0a7cb8f4000f0474aa44d21c2e4917( int␣
↪value ){
    int native_m[2][3];
    boost::python::list py_m;
    ::ft::filler(native_m, value);
    for (int row = 0; row < 2; ++row ){
        boost::python::list pyrow;
```

```
        pyplus_conv::copy_container( native_m[row]
                                     , native_m[row] + 3
                                     , pyplus_conv::list_inserter( pyrow ) );
        py_m.append( pyrow );
    }
    return bp::object( py_m );
}

BOOST_PYTHON_MODULE(...){
    ...
   typedef boost::python::object ( *filler_function_type )( int );

   bp::def(
       "filler"
       , filler_function_type( &filler_7b0a7cb8f4000f0474aa44d21c2e4917 )
       , ( bp::arg("value") ) );
}
```

`inout_static_matrix` transformer

Definition

`inout_static_matrix` transformer is a combination of *input* and *output* transformers. It allows one to call a C++ function, which takes 2D array using Python `list` class

"input_static_matrix" transformer takes as first argument name or index of the original function argument. The argument should have "array" or "pointer" type. The second and the third arguments specify rows and columns sizes.

Limitations

This transformer could not be applied on virtual functions.

Example

```
int sum_and_fill( int m[2][3], int value ){
    int result = 0;
    for( int r = 0; r < 2; ++r ){
        for( int c = 0; c < 3; ++c ){
            result += m[r][c];
            m[r][c] *= value;
        }
    }
    return result;
}
```

In order to expose `sum_and_fill` function we need to create a small wrapper. The following *Py++* code does it for you:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder.module_builder_t( ... )
```

```
sum_and_fill = mb.free_fun( 'sum_and_fill' )
sum_and_fill.add_transformation( ft.inout_static_matrix('m', rows=2,␣
↪columns=3) )
```

What you see below is the relevant pieces of generated code:

```
static boost::python::tuple sum_and_fill_ec4892ec81f672fe151a0a2caa3215f4(␣
↪boost::python::object m, int value ){
    int native_m[2][3];
    boost::python::list py_m;
    pyplus_conv::ensure_uniform_sequence< boost::python::list >( m, 2 );
    for( size_t row = 0; row < 2; ++row ){
        pyplus_conv::ensure_uniform_sequence< int >( m[row], 3 );
        pyplus_conv::copy_sequence( m[row], pyplus_conv::array_inserter(␣
↪native_m[row], 3 ) );
    }
    int result = ::ft::sum_and_fill(native_m, value);
    for (int row2 = 0; row2 < 2; ++row2 ){
        boost::python::list pyrow;
        pyplus_conv::copy_container( native_m[row2]
                                     , native_m[row2] + 3
                                     , pyplus_conv::list_inserter( pyrow ) );
        py_m.append( pyrow );
    }
    return bp::make_tuple( result, py_m );
}

BOOST_PYTHON_MODULE(ft_inout_static_matrix){
    { //::ft::sum_and_fill

        typedef boost::python::tuple ( *sum_and_fill_function_type )(␣
↪boost::python::object,int );

        bp::def(
            "sum_and_fill"
            , sum_and_fill_function_type( &sum_and_fill_
↪ec4892ec81f672fe151a0a2caa3215f4 )
            , ( bp::arg("m"), bp::arg("value") ) );

    }
}
```

The set doesn't cover all common use cases, but it will grow with every new version of *Py++*. If you created your own transformer consider to contribute it to the project.

I suggest you to start reading *output* transformer. It is pretty simple and well explained.

All built-in transformers could be applied on any function, except constructors and pure virtual functions. The support for them be added in future releases.

You don't have to worry about call policies. You can set the call policy and *Py++* will generate the correct code.

You don't have to worry about the number of arguments, transformers or return value. *Py++* handles pretty well such use cases.

Default arguments

Introduction

There is more than one way to export function with default arguments. Before we proceed, please take a look on the following class:

```
struct X
{
    bool f(int a=12)
    {
        return true;
    }
};
```

Do nothing approach

By default *Py++* exposes function with its default arguments.

```
namespace bp = boost::python;

BOOST_PYTHON_MODULE(pyplusplus){
  bp::class_< X >( "X" )
      .def(
          "f"
          , &::X::f
          , ( bp::arg("a")=(int)(12) ) );
}
```

The additional value of the approach is keyword arguments. You will be able to call function `f` like this:

```
x = X()
x.f( a=13 )
```

Default values, using macros

`BOOST_PYTHON_FUNCTION_OVERLOADS` and `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` macros can help to deal with default values too. You can turn `use_overload_macro` to `True`:

```
import module_builder

mb = module_builder.module_builder_t( ... )
x = mb.class_( "X" )
x.member_function( "f" ).use_overload_macro = True
#-----------------------^^^^^^^^^^^^^^^^^^^^^^^^^
```

Registration order problem

There is different trades-off between these approaches. In general you should use the first one, until you have "registration order" problem:

```
struct S1;
struct S2;

struct S1{
    void do_smth( S2* s2=0 );
};

struct S2{
    void do_smth( S1 s1=S1() );
};

BOOST_PYTHON_MODULE( ... ){
    using namespace boost::python;

    class_< S2 >( "S2" )
        .def( "do_smth", &S2::do_smth, ( arg("s1")=S1() ) );

    class_< S1 >( "S1" )
        .def( "do_smth", &S1::do_smth, ( arg("s2")=object() ) );

}
```

The good news is that it is very easy to identify the problem: the module could not be loaded. The main reason is that expression `arg("s1")=S1()` requires `S1` struct to be registered. CastXML reports default arguments as strings. *Py++* doesn't have enough information to generate code with the right class registration order. In this case you have to instruct *Py++* to use macros:

```
import module_builder

mb = module_builder.module_builder_t( ... )
s2 = mb.class_( "S2" )
s2.member_function( "do_smth" ).use_overload_macro = True
```

When you switch to macros, than:

- You will not be able to override virtual functions in Python.
- You will not be able to use "named" arguments.
- You will not be able to set the functions documentation.

Special case

Class constructors are special case:

```
struct S1;
struct S2;

struct S1{
    S1( S2* s2=0 );
};

struct S2{
    S2( S1 s1=S1() );
};
```

You cannot use same work around and *Py++* (version 0.8.2) could not help you. The use case presented here is a little

bit esoteric. If you have such use case and you cannot change the source code, consider contacting *Py++* developers. I am sure we will be able to help you.

make_constructor

Introduction

Boost.Python allows us to register some function as Python class `__init__` method. This could be done using make_constructor functionality.

Not every function could be registered as `__init__` method. The function return type should be a pointer or a smart pointer to the new class instance.

Usage example

I am going to use the following code to demonstrate the functionality:

```
#include <memory>

namespace mc{

struct number_t{

    static std::auto_ptr<number_t> create( int i, int j);

    int x;
};

std::auto_ptr<number_t> create(int i);

}//namespace mc
```

The code is pretty simple - it defines two `create` functions, which construct new class `number_t` instances.

Py++ configuration is pretty simple:

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t( ... )
mc = mb.namespace( 'mc ')
number = mc.class_( 'number_t' )
number.add_fake_constructors( mc.calldefs( 'create' ) )
#------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Basically you associate with the class the functions, you want to register as the class `__init__` method.

The method `add_fake_constructors` takes as argument a reference to "create" function or a list of such.

The generated code is pretty boring and the only thing I would like to mention is that the function will **not** be exposed as a standalone function.

The usage code is even more boring:

```
from your_module import number_t

number = number_t( 1 )
print number.x
```

```
number = number_t( 1, 2 )
print number.x
```

Overloading

Introduction

Things get a little bit complex, when you have to export overloaded functions. In general the solution is to explicitly say to compiler what function you want to export, by specifying its type. Before we proceed, please take a look on the following class:

```
struct X
{
    bool f(int a)
    {
        return true;
    }

    bool f(int a, double b)
    {
        return true;
    }

    bool f(int a, double b, char c)
    {
        return true;
    }
};
```

This class has been taken from Boost.Python tutorials.

There are few approaches, which you can use in order to export the functions.

Do nothing approach

I am sure you will like "do nothing" approach. *Py++* recognize that you want to export an overloaded function and will generate the right code:

```
namespace bp = boost::python;

BOOST_PYTHON_MODULE(pyplusplus){
  bp::class_< X >( "X" )
      .def(
          "f"
          , (bool ( ::X::* )( int ) )( &::X::f )
          , ( bp::arg("a") ) )
      .def(
          "f"
          , (bool ( ::X::* )( int,double ) )( &::X::f )
          , ( bp::arg("a"), bp::arg("b") ) )
      .def(
          "f"
          , (bool ( ::X::* )( int,double,char ) )( &::X::f )
```

```
          , ( bp::arg("a"), bp::arg("b"), bp::arg("c") ) );
}
```

"create_with_signature" approach

Well, while previous approach is very attractive it does not work in all cases and have a weakness.

Overloaded template function

I am sure you already know the following fact, but still I want to remind it:

- CastXML doesn't report about un-instantiated templates

It is very important to understand it. Lets take a look on the following source code:

```
struct Y{

    void do_smth( int );

    template< class T>
    void do_smth( T t );

};
```

If you didn't instantiate(use) `do_smth` member function, than CastXML will not report it. As a result, *Py++* will not be aware of the fact that `do_smth` is an overloaded function. To make the long story short, the generated code will not compile. You have to instruct *Py++* to generate code, which contains function type:

```
from pyplusplus import module_builder

mb = module_builder.module_builder_t( ... )
y = mb.class_( 'Y' )
y.member_function( 'do_smth' ).create_with_signature = True
#______________________________^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Every *Py++* class, which describes C++ function\operator has `create_with_signature` property. You have to set it to `True`. Default value of the property is computed. If the exported function is overloaded, then its value is `True` otherwise it will be `False`.

Do nothing approach weakness

Code modification - the weakness of the "do nothing" approach. We live in the dynamic world. You can create bindings for a project, but a month letter, the project developers will add a new function to the exported class. Lets assume that the new function will introduce overloading. If `create_with_signature` has `False` as a value, than the previously generated code will not compile and you will have to run code generator one more time.

Consider to explicitly set `create_with_signature` to `True`. It will save your and your users time in future.

```
mb = module_builder_t( ... )
mb.calldefs().create_with_signature = True
```

Overloading using macros

Boost.Python provides two macros, which help you to deal with overloaded functions:

- `BOOST_PYTHON_FUNCTION_OVERLOADS`
- `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS`

Boost.Python tutorials contain an explanation about this macros.

You can instruct *Py++* to generate code, which will use the macros:

```
import module_builder

mb = module_builder.module_builder_t( ... )
x = mb.class_( "X" )
x.member_functions( "f" ).use_overload_macro = True
#------------------------^^^^^^^^^^^^^^^^^^^^^^^^^
```

Member and free functions declaration classes have `use_overload_macro` property. The default value of the property is `False`.

You don't really have to use the macros, unless you have "registration order" problem. The problem and work around described in *default arguments* document.

Registration order

Introduction

"... I would very much like to pass booleans from Python to C++ and have them accepted as boils. However I cannot seem to do this. ..."

"... My class has 2 "append" functions. The first one, has single argument with type "const char*", the second one also has single argument with type "char". It seems, that I am not able to call the first function. ..."

If you have problem similar to described ones, than I am almost sure you have registration order problem.

Example

```
struct tester_t{
    tester_t() {}

    const char* append(const char*)
    { return "append(const char *)"; }

    const char* append(const char)
    { return "append(const char)"; }

    const char* do_smth( bool )
    { return "do_smth( bool )"; }

    const char* do_smth( int )
    { return "do_smth( int )"; }
};
```

Py++ generates code, that register functions in the order they appear in the source code:

```
namespace bp = boost::python;

BOOST_PYTHON_MODULE(my_module){
  bp::class_< tester_t >( "tester_t" )
      .def( bp::init< >() )
      .def( "append"
          , (char const * ( ::tester_t::* )( char const * ) )( &::tester_t::append ) )
      .def( "append"
          , (char const * ( ::tester_t::* )( char const ) )( &::tester_t::append ) )
      .def( "do_smth"
          , (char const * ( ::tester_t::* )( bool ) )( &::tester_t::do_smth ) )
      .def( "do_smth"
          , (char const * ( ::tester_t::* )( int ) )( &::tester_t::do_smth ) );
}
```

Registration order pitfalls

Do you want to guess what is the output of the following program:

```
import my_module
tester = my_module.tester_t()
print tester.do_smth( True )
print tester.do_smth( 10 )
print tester.append( "Hello world!" )
```

?

The output is:

do_smth(int)

do_smth(int)

append(const char)

Unexpected, right? The registration order of exposed overloaded functions is important. Boost.Python tries overloads in reverse order of definition.

If I understand right, Boost.Python tries to match in reverse order the overloaded functions, if it can convert Python arguments to C++ ones, it does this and calls the function.

Now, when you understand the behavior, it should be pretty simple to provide a correct functionality:

1. You can change alias of the function, by mangling the type of the argument into it:

   ```
   mb = module_builder_t( ... )
   for f in mb.class_( 'tester_t' ).member_functions():
       f.alias = f.alias + f.arguments[0].type.decl_string
   ```

2. You can reorder the declarations within the source file.

3. You can ask *Py++* to generate code, which takes into account the order of declarations:

   ```
   from pyplusplus.creators_factory import sort_algorithms

   sort_algorithms.USE_CALLDEF_ORGANIZER = True
   # The functionality is available from version 0.8.3
   ```

4. The last and the perfect solution. *Py++* will let you know, when your code has such problem. The functionality is available from version 0.8.3. After this you can change the aliases of the functions. The third step is to create small "dispatch" function in Python:

```
import my_module

def tester_t_do_smth( self, value ):
    if isinstance( value, bool ):
        self.do_smth_bool( value ):
    else:
        self.do_smth_int( value )

tester_t.do_smth = tester_t_do_smth
```

The technique shown here described pretty good in Boost.Python Extending Wrapped Objects in Python tutorials .

May be in future, *Py++* will generate this code for you. Anyway, if you have a lot of use cases like this consider to generate Python code and not to write it manually.

4.1.11 Getting started

- *Boost.Python tutorials*.
- *C & ctypes tutorials*.
- *D language & C tutorials*.

4.2 Users and quotes

4.2.1 What do they say?

"... If you can, use pyplusplus over pyste. I say that for ALL users of pyste, pyplusplus is now mature enough to be useful as well as being actively developed. It can also do quite a few tricks pyste cannot. "

Niall Douglas, the author of TnFOX library

"... On a related note, I highly suggest that any users out there that have tried/used Pyste but have found it to be too lacking in power should really give pyplusplus a try. It has allowed me to do everything I ever wanted to do with Pyste and couldn't and then some. It is really a great tool and I can't thank Roman enough for taking the time to create it and make it available. "

Allen Bierbaum, the author of PyOpenSG library

"... This rule based approach is amazing for maintenance, as it reduces the turnaround for binding new code. If the new Ogre API's follow similar rules and standards as previously defined, the same set of rules will appropriately bind the new API without any effort on the part of the maintainers. "

" ... In general, I've really liked working with pyplusplus. I've probably spent 20-30 hours working on these bindings, and they are very close to being equivalent to the PyOgre bindings (when I last used them). "

Lakin Wecker, the author of Python-OGRE project

"... Py++ allows the wrappers to be "automagically" created, which means it's much easier to keep things up to date (the maintenance on the Py++ based wrapper is tiny compared to any other system I've used). It also allows us to wrap other libraries fairly easily. "

Andy Miller, a developer of Python-OGRE project

"... I tried Py++ and it indeed automatically handles the case I outlined above concerning C-array members, and with much less tedious writing of registration code. I also found it convenient to use to insert some other C++ code for each of my structures that normally I wrote by hand. The API docs and examples on your webpage were very helpful. "

David Carpman

"... I started a few months ago to develop a set of Python bindings for OpenCascade modeling/visualization library. After a quick tour to evaluate different solutions, my choice lead me to Py++, which is a very convenient tool : I was able to achieve the first release of my project only two weeks after the project start !"

Paviot Thomas

4.2.2 Who is using Py++?

- European Space Agency - ReSP project

  ReSP is an Open-Source hardware simulation platform targeted for multiprocessor systems. ReSP will provide a framework for composing a system by connecting components chosen from a given repository or developed by the designer. ReSP will provide also also a framework for fault injection campaigns for the analysis of the reliability level of the system.

  ReSP engineers are developing the simulator core in Python language for exploiting reflective capabilities (missing in a pure C++ environment) that can be exploited for connecting components in a dynamic way and for enabling non-intrusive fault injection activity. Components will be described in SystemC and TLM libraries that are high level hardware description languages based on C++.

- Allen Bierbaum, the author of PyOpenSG project, is using *Py++* to create Python bindings for OpenSG

  OpenSG - is a portable scene graph system to create realtime graphics programs, e.g. for virtual reality applications.

- Matthias Baas, the author of Python Computer Graphics Kit project, is using *Py++* to create Python bindings for Maya C++ SDK.

- Lakin Wecker, the author of Python-OGRE project, is using *Py++* to create Python bindings for OGRE.

  OGRE - is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics.

- Andy Miller, another developer of Python-OGRE project, is using *Py++* to create Python bindings for:
  - CEGUI - a free library providing windowing and widgets for graphics APIs / engines where such functionality is not available, or severely lacking.
  - ODE - an open source, high performance library for simulating rigid body dynamics.
  - OIS - an object oriented input system.
  - All in all, Python-OGRE project contains bindings for more than 30 libraries. You can find code generation scripts here: https://python-ogre.svn.sourceforge.net/svnroot/python-ogre/trunk/python-ogre/code_generators/

- Rising Sun Pictures company is using *Py++* to create Python bindings for Apple Shake API. PyShake enables running of Python code from within Shake and by exposing the Shake API to Python.

- Paviot Thomas, the author of pythonOCC project, is using *Py++* to create Python bindings for OpenCascade, a 3D modeling & numerical simulation library.

- Adrien Saladin, the author of PTools project, is using *Py++* to create an opensource molecular docking library.

- I am :-). I created Python bindings for the following libraries:
  - Boost.Date_Time

- Boost.CRC
- Boost.Rational
- Boost.Random

You can download the bindings from https://sourceforge.net/project/showfiles.php?group_id=118209 .

4.3 Download & Install

4.3.1 Py++ on SourceForge

Py++ project is hosted on SourceForge. Using SourceForge services you can:

1. get access to source code
2. get access to latest release version of *Py++*

4.3.2 Subversion access

http://sourceforge.net/svn/?group_id=118209

4.3.3 Download

https://sourceforge.net/project/showfiles.php?group_id=118209

4.3.4 Installation

In command prompt or shell change current directory to be "pyplusplus-X.Y.Z". "X.Y.Z" is version of *Py++*. Type the following command:

```
python setup.py install
```

After this command complete, you should have installed *Py++* package.

Boost.Python installation

Users of Microsoft Windows can enjoy from simple installer for Boost Libraries. You can find it here. Take a look on new getting started guide for Boost libraries.

Another very valuable link related to Boost is http://engineering.meta-comm.com/boost.aspx . You will find hourly snapshots of the source code and the documentation for all Boost libraries.

4.3.5 Dependencies

- pygccxml

4.4 Examples

4.4.1 unittests

Py++ has more then 110 tests and the number is growing. Consider to take a look on them. The tests names are descriptive, so it should be not too difficult to find the desired functionality.

4.4.2 "pyplusplus/examples" directory

The directory contains few well documented examples:

- "pycrc" - Boost.Python based bindings for Boost.CRC library
- "custom_code_creator" - Boost.Python based bindings, which explains how to build and integrate the user defined code creator. The example exposes a class "get/set" member functions as Python properties
- "pygmplib" - `ctypes` based bindings for GMP library.

In order to see the geneated code and/or try it, execute "scons" in that directory.

4.5 Architecture

4.5.1 Introduction

This document will describe an architecture behind *Py++*.

4.5.2 Py++ & pygccxml integration

C++

C++ is very powerful programming language. The power brings complexity. It is not an easy task to parse C++ source files and to create in memory representation of declarations tree. The declarations tree is worth nothing, if a user is not able to explorer it, to run queries against it or to find out traits of a declaration or a type.

On the earlier stage of the development, I realized, that all this functionality does not belong to code generator and should be implemented out side of it. pygccxml project was born. pygccxml made the code generator to be smaller and C++ parser independent. It provides the following services:

- definition of classes, that describe C++ declaration and types, and their analyzers (type traits)
- C++ source files parsing and caching functionality

Py++ uses those services to:

- extract declarations from source files and to provide powerful query interface
- find out a declaration default configuration:
  - call policies for functions
  - indexing suite parameters
  - generate warnings/hints
  - ...

Integration details

Py++ uses different approaches to expose these services to the user.

Parsing integration

Py++ provides it's own "API" to configure pygccxml parsing services. The "API" I am talking about, is arguments to `module_builder.__init__` method. We think, that exposing those services via *Py++* simplifies its usage.

Declarations tree integration

Declarations tree API consists from 3 parts:

- interface definition:
  - `declaration_t` and all classes that derive from it
  - `type_t` and all classes that derive from it
- type traits
- query engine API

The user should be familiar with these parts and relevant API. In my opinion, wrapping or hiding the API will not provide an additional value. The interface of all those services is pretty simple and well polished.

Before I explain how these services are integrated, take a look on the following source code:

```
mb = module_builder_t( ... )

details = mb.namespace( 'details' )
details.exclude()

my_class = mb.class_( 'my_class' )
my_class.rename("MyClass")
```

What you see here, is a common pattern, that will appear in all projects, that use *Py++*:

- find the declaration(s)
- give the instruction(s) to the code generator engine

What is the point of this example? From the user point of view it is perfectly good, it makes a lot of sense to configure the code generation engine, using the declarations tree. How does *Py++* add missing functionality to `pygccxml.declarations` classes? There were few possible solutions to the problem. The following one was implemented:

1. `pygccxml.parser` package interface was extended. Instead of creating a concrete instance of declaration classes, `pygccxml.parser` package uses a factory.
2. `pyplusplus.decl_wrappers` package defines classes, which derive from `pygccxml.declarations` classes and defines the factory.

The implemented solution is not the simplest one, but it provides an additional value to the project:

- the code generation engine configuration and declarations tree are tightly coupled
- the functionality provided by `pygccxml.declarations` and `pygccxml.parser` packages is available for `pyplusplus.decl_wrappers` classes
- classes defined in `pyplusplus.decl_wrappers` package implement the following functionality:

  - setting reasonable defaults for the code generation engine(call policies, indexing suite, ...)
  - provides user with additional information(warnings and hints)
- as a bonus, pygccxml remained to be stand-alone project

4.5.3 Code generation engine

Code generation for Boost.Python library is a difficult process. There are two different problems the engine should solve:

- What code should be created in order to export a declaration?
- How it should be written to files?

  Remember, *Py++* is targeting big projects. It cannot generate all code in one file - this will not work, not at all.

`Code creators` and file writers provides solution for both problems.

`Code creators`

Do you know how many ways exist to export member function? If you will try to answer the question, consider the following function characteristics and their mix:

- virtuality(non virtual, virtual or pure virtual)
- access level(public, protected or private)
- static\non static
- overloads

As you see, there are a lot of use cases. How do `code creators` solve the problem?

Definition

`Code creator` is an in-memory fragment of a C++ code.

Also, `code creator` can represent an arbitrary C++ code, in practice it represents logically complete block.

Example of `code creators`:

- `code_creators.enum_t` generates registration code for an enumeration
- `code_creators.mem_fun_pv_t` generates registration code for public, pure virtual function
- `code_creators.mem_fun_pv_wrapper_t` generates declaration code for public, pure virtual function
- `code_creators.include_t` generates include directives
- `code_creators.custom_text_t` adds some custom(read user) text\code to the generated code

There are primary two groups of `code creators`: declaration based and others.

Declaration based `code creator` keeps reference to the declaration (`pyplusplus.decl_wrapper.*` class instance). During code generation process, it reads its settings(the code generation engine instructions) from the declaration. Declaration based `code creators` also divided into two groups. The first group creates registration code, where the second one creates wrapper\helper declaration code.

I will reuse this example, from Boost.Python tutorials.

1. `BaseWrap::f`, `BaseWrap::default_f` - declaration code is created by `code_creators.mem_fun_v_wrapper_t`
2. `f` registration code is created by `code_creators.mem_fun_v_t`. This code creator also keeps reference to the relevant instance of `code_creators.mem_fun_v_wrapper_t` class.

Composite `code creator` is a creator, which contains other creators. Composite `code creator` embeds the code, created by internal `code creators`, within the code it creates. For example:

- `code_creators.class_t`:

  First of all it creates class registration code (`class_<...>`), after this it appends to it code generated by internal creators.

- `code_creators.module_body_t`:

  Here is "cut & paste" of the relevant code from the source file:

```
def _create_impl(self):
    result = []
    result.append( "BOOST_PYTHON_MODULE(%s){" % self.name )
    result.append( compound.compound_t.create_internal_code( self.creators ) )
    result.append( "}" )
    return os.linesep.join( result )
```

Code creators tree

`code_creators.module_t` class is a top level `code creator`. Take a look on the following possible "snapshot" of the `code creators tree`:

```
<module_t ...>
    <license_t ...>
    <include_t ...>
    <include_t ...>
    <class_wrapper_t ...>
        <mem_fun_v_wrapper_t ...>
        <mem_fun_v_wrapper_t ...>
    <module_body_t ...>
        <enum_t ...>
        <class_t ...>
            <mem_fun_v_t ...>
            <member_variable_t ...>
        <free_function_t ...>
        <...>
```

You can think about `code creators tree` as some kind of AST.

Code creators tree construction

`pyplusplus.creators_factory` package is responsible for the tree construction. `pyplusplus.creators_factory.creator_t` is the main class of the package. It creates the tree in few steps:

1. It builds set of exposed declarations.
2. It sort the set. Boost.Python has few rules, that forces the user to export a declaration before another one.
3. It creates `code creators` and put them into the right place within the tree.
4. If a declaration describes C++ class, it applies these steps to it.

Another responsibility of `creator_t` class, is to analyze declarations and their dependency graphs. As a result, this class can:

- find out a class `HeldType`
- find out smart pointers conversion, which should be registered
- find out STD containers, which should be exported
- warn user, if some declaration is not exported and it used somewhere in exported declarations (**not implemented**)

`File writers`

`File writers` classes are responsible for writing `code creators tree` into the files. *Py++* implements the following strategies of writing `code creators tree` into files:

- single file
- multiple files - provides a solution to compilation time and memory usage problem
- multiple files, with huge classes are written into multiple files - provides a solution for compiler limit problem.

The more sophisticated approach, the better understanding of `code creators` is required from the `file writers`.

4.5.4 `module_builder` package

This package provides an interface to all code generator engine services.

4.5.5 Conclusion

It safe to use *Py++* for big and small projects!

4.6 Boost.Python - lessons learned

4.6.1 Preamble

Software development is an interactive process. During *Py++* development I see many interesting problems and even more interesting solutions.

On this page you will find my collection of the solutions to some of the problems.

Easy extending guide

Introduction

"... Boost.Python library is designed to wrap C++ interfaces non-intrusively, so that you should not have to change the C++ code at all in order to wrap it."

The previous statement is almost true. There are few use cases that the library doesn't support. This guide will list some of them and will offer few possible solutions.

Pointer to function

Boost.Python doesn't handle "pointer to function" functionality. You cannot pass it as function argument or keep it, as a member variable.

The simple work-around is to use command design pattern

Problematic function arguments types

C arrays

Boost.Python doesn't handle `C arrays`, the only exception are `char*` and `wchar_t*`.

Consider the following function:

```
int write( int* data, size_t size );
```

The technical reasons are not the only one that prevent Boost.Python to expose such functions, there is a mental one: such interface is not intuitive for Python developers. They expect to pass single argument. For example, built-in `file.write` method takes a single argument - sequence of characters.

Work-around:

1. With small help from the developer, Py++ generates code which feets well into Python developer mental model. Pure virtual member functions are a special case, which Py++ doesn't handle right now.
2. Use STL containers, `std::vector<...>` and others.

Immutable by reference

Python defines few fundamental types as "immutable". The value of an instance of the immutable type could not be changed after construction. Try to avoid passing the immutable types by reference.

Immutable types:

- `char`
- `signed char`
- `unsigned char`
- `wchar_t`
- `short int`
- `short unsigned int`
- `bool`
- `int`
- `unsigned int`
- `long int`
- `long unsigned int`
- `long long int`
- `long long unsigned int`
- `float`

- `double`
- `long double`
- `complex double`
- `complex long double`
- `complex float`
- `std::string`
- `std::wstring`
- C++ `enum` is mapped to Python `int` type
- smart pointers

Work around:

- Just don't pass them by reference :-)
- With small help from the developer, Py++ generates code which work-arounds this issue, but the resulting interface is ugly.

void*

In most cases, `void*` is used when a developer has to deal with a memory block. Python provides support for this functionality, but I still didn't find an easy and intuitive way to expose it. There is no work-around for this issue.

If you use `void*` to pass a reference to some object, than Boost.Python and Py++ support such use case.

Memory managment

- Use `std::auto_ptr` to transfer ownership and responsibility for an object destruction.
- The only well supported smart pointer class is `boost::shared_ptr`. I suggest you to use it all the time, especially in cases where you want to create object from Python and pass ownership to C++ code. You don't want the headache associated with this task.

How to expose custom smart pointer?

Files

User defined "smart pointer" class

```
#ifndef smart_ptr_t_19_09_2006
#define smart_ptr_t_19_09_2006

#include <assert.h>

//The smart_ptr_t class has been created based on Ogre::SharedPtr class
//http://www.ogre3d.org/docs/api/html/OgreSharedPtr_8h-source.html

namespace smart_pointers{

template<class T>
class smart_ptr_t {
```

```
protected:
    T* m_managed;
    unsigned int* m_use_count;
public:

    smart_ptr_t()
    : m_managed(0), m_use_count(0)
    {}

    //rep should not be NULL
    explicit smart_ptr_t(T* rep)
    : m_managed(rep), m_use_count( new unsigned int(1) )
    {}

    //Every custom smart pointer class should have copy constructor and
    //assignment operator. Probably your smart pointer class already has this
    //functionality.

    smart_ptr_t(const smart_ptr_t& r)
    : m_managed(0), m_use_count(0)
    {
        m_managed = r.get();
        m_use_count = r.use_count_ptr();
        if(m_use_count){
            ++(*m_use_count);
        }
    }

    smart_ptr_t& operator=(const smart_ptr_t& r){
        if( m_managed == r.m_managed ){
            return *this;
        }
        release();

        m_managed = r.get();
        m_use_count = r.use_count_ptr();
        if(m_use_count){
            ++(*m_use_count);
        }
        return *this;
    }

    //Next two functions allow one to construct smart pointer from an existing one,
    //which manages object with a different type.
    //For example:
    //    struct base{...};
    //    struct derived : base { ... };
    //    smart_ptr_t< base > b( smart_ptr_t<derived>() );
    //
    //This functionality is very important for C++ Python bindings. It will allow
    //you to register smart pointer conversion:
    //    boost::python::implicitly_convertible< smart_ptr_t< derived >, smart_ptr_t<
↪base > >();
    template<class Y>
    smart_ptr_t(const smart_ptr_t<Y>& r)
    : m_managed(0), m_use_count(0)
    {
        m_managed = r.get();
```

```
        m_use_count = r.use_count_ptr();
        if(m_use_count){
            ++(*m_use_count);
        }
    }

    template< class Y>
    smart_ptr_t& operator=(const smart_ptr_t<Y>& r){
        if( m_managed == r.m_managed ){
            return *this;
        }
        release();

        m_managed = r.get();
        m_use_count = r.use_count_ptr();
        if(m_use_count){
            ++(*m_use_count);
        }
        return *this;
    }

    virtual ~smart_ptr_t() {
        release();
    }

    inline T& operator*() const {
        assert(m_managed); return *m_managed;
    }

    inline T* operator->() const {
        assert(m_managed); return m_managed;
    }

    inline T* get() const {
        return m_managed;
    }

    inline unsigned int* use_count_ptr() const {
        return m_use_count;
    }

protected:

    inline void release(void){
        bool destroy_this = false;

        if( m_use_count ){
            if( --(*m_use_count) == 0){
                destroy_this = true;
            }
        }
        if (destroy_this){
            destroy();
        }
    }

    virtual void destroy(void){
        delete m_managed;
```

```
        delete m_use_count;
    }
};

} //smart_pointers

#endif //smart_ptr_t_19_09_2006
```

To be exposed C++ classes

```
#ifndef classes_11_11_2006
#define classes_11_11_2006

#include "smart_ptr.h"

struct base_i{
public:
    virtual ~base_i() {}
    virtual int get_value() const = 0;
};

struct derived_t : base_i{
    derived_t(){}
    virtual int get_value() const{ return 0xD; }
};

// Some smart pointer classes does not have reach interface as boost ones.
// In order to provide same level of convenience, users are forced to create
// classes, which derive from smart pointer class.
struct derived_ptr_t : public smart_pointers::smart_ptr_t< derived_t >{

    derived_ptr_t()
    : smart_pointers::smart_ptr_t< derived_t >()
    {}

    explicit derived_ptr_t(derived_t* rep)
    : smart_pointers::smart_ptr_t<derived_t>(rep)
    {}

    derived_ptr_t(const derived_ptr_t& r)
    : smart_pointers::smart_ptr_t<derived_t>(r) {}

    derived_ptr_t( const smart_pointers::smart_ptr_t< base_i >& r)
    : smart_pointers::smart_ptr_t<derived_t>()
    {
        m_managed = static_cast<derived_t*>(r.get());
        m_use_count = r.use_count_ptr();
        if (m_use_count)
        {
            ++(*m_use_count);
        }
    }

    derived_ptr_t& operator=(const smart_pointers::smart_ptr_t< base_i >& r)
    {
        if (m_managed == static_cast<derived_t*>(r.get()))
```

```
            return *this;
        release();
        m_managed = static_cast<derived_t*>(r.get());
        m_use_count = r.use_count_ptr();
        if (m_use_count)
        {
            ++(*m_use_count);
        }

        return *this;
    }
};

// Few functions that will be used to test custom smart pointer functionality
// from Python.
derived_ptr_t create_derived(){
    return derived_ptr_t( new derived_t() );
}

smart_pointers::smart_ptr_t< base_i > create_base(){
    return smart_pointers::smart_ptr_t< base_i >( new derived_t() );
}


// Next function could be exposed, but it could not be called from Python, when
// the argument is the instance of a derived class.
//
// This is the explanation David Abrahams gave:
//   Naturally; there is no instance of smart_pointers::smart_ptr_t<base_i> anywhere␣
↪in the
//   Python object for the reference to bind to. The rules are the same as in C++:
//
//   int f(smart_pointers::smart_ptr_t<base>& x);
//   smart_pointers::smart_ptr_t<derived> y;
//   int z = f(y);               // fails to compile

inline int
ref_get_value( smart_pointers::smart_ptr_t< base_i >& a ){
    return a->get_value();
}

inline int
val_get_value( smart_pointers::smart_ptr_t< base_i > a ){
    return a->get_value();
}

inline int
const_ref_get_value( const smart_pointers::smart_ptr_t< base_i >& a ){
    return a->get_value();
}


struct numeric_t{
    numeric_t()
    : value(0)
    {}

    int value;
```

```
};

smart_pointers::smart_ptr_t< numeric_t > create_numeric( int value ){
    smart_pointers::smart_ptr_t< numeric_t > num( new numeric_t() );
    num->value = value;
    return num;
}

int get_numeric_value( smart_pointers::smart_ptr_t< numeric_t > n ){
    if( n.get() ){
        return n->value;
    }
    else{
        return 0;
    }
}



namespace shared_data{

// Boost.Python has small problem with user defined smart pointers and public
// member variables, exposed using def_readonly, def_readwrite functionality
// Read carefully "make_getter" documentation.
// http://boost.org/libs/python/doc/v2/data_members.html#make_getter-spec
// bindings.cpp contains solution to the problem.

struct buffer_t{
    buffer_t() : size(0) {}
    int size;
};

struct buffer_holder_t{
    buffer_holder_t()
    : data( new buffer_t() )
    {}

    smart_pointers::smart_ptr_t< buffer_t > get_data(){ return data; }

    smart_pointers::smart_ptr_t< buffer_t > data;
};

}

#endif//classes_11_11_2006
```

C++/Python bindings code

```
#include "boost/python.hpp"
#include "classes.hpp"

namespace bp = boost::python;

namespace smart_pointers{
    // "get_pointer" function returns pointer to the object managed by smart pointer
    // class instance
```

```
    template<class T>
    inline T * get_pointer(smart_pointers::smart_ptr_t<T> const& p){
        return p.get();
    }

    inline derived_t * get_pointer(derived_ptr_t const& p){
        return p.get();
    }
}

namespace boost{ namespace python{

    using boost::get_pointer;

    // "pointee" class tells Boost.Python the type of the object managed by smart
    // pointer class.
    // You can read more about "pointee" class here:
    // http://boost.org/libs/python/doc/v2/pointee.html

    template <class T>
    struct pointee< smart_pointers::smart_ptr_t<T> >{
        typedef T type;
    };

    template<>
    struct pointee< derived_ptr_t >{
        typedef derived_t type;
    };

} }


// "get_pointer" and "pointee" are needed, in order to allow Boost.Python to
// work with user defined smart pointer

struct base_wrapper_t : base_i, bp::wrapper< base_i > {

    base_wrapper_t()
    : base_i(), bp::wrapper< base_i >()
    {}

    virtual int get_value(  ) const {
        bp::override func_get_value = this->get_override( "get_value" );
        return func_get_value(  );
    }

};

struct derived_wrapper_t : derived_t, bp::wrapper< derived_t > {

    derived_wrapper_t()
    : derived_t(), bp::wrapper< derived_t >()
    {}

    derived_wrapper_t(const derived_t& d)
    : derived_t(d), bp::wrapper< derived_t >()
    {}
```

```
    derived_wrapper_t(const derived_wrapper_t&)
    : derived_t(), bp::wrapper< derived_t >()
    {}

    virtual int get_value() const  {
        if( bp::override func_get_value = this->get_override( "get_value" ) )
            return func_get_value(  );
        else
            return derived_t::get_value(  );
    }

    int default_get_value() const  {
        return derived_t::get_value( );
    }

};

BOOST_PYTHON_MODULE( custom_sptr ){
    bp::class_< base_wrapper_t
                , boost::noncopyable
                , smart_pointers::smart_ptr_t< base_wrapper_t > >( "base_i" )
    // -----------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    // HeldType of the abstract class, which is managed by custom smart pointer
    // should be smart_pointers::smart_ptr_t< base_wrapper_t >.
        .def( "get_value", bp::pure_virtual( &base_i::get_value ) );

    // Register implicit conversion between smart pointers. Boost.Python library
    // can not discover relationship between classes. You have to tell about the
    // relationship to it. This will allow Boost.Python to treat right, the
    // functions, which expect to get as argument smart_pointers::smart_ptr_t< base_i␣
↪> class
    // instance, when smart_pointers::smart_ptr_t< derived from base_i > class␣
↪instance is passed.
    //
    // For more information about implicitly_convertible see the documentation:
    // http://boost.org/libs/python/doc/v2/implicit.html .
    bp::implicitly_convertible<
                  smart_pointers::smart_ptr_t< base_wrapper_t >
                , smart_pointers::smart_ptr_t< base_i > >();

    // The register_ptr_to_python functionality is explaned very well in the
    // documentation:
    // http://boost.org/libs/python/doc/v2/register_ptr_to_python.html .
    bp::register_ptr_to_python< smart_pointers::smart_ptr_t< base_i > >();

    bp::class_< derived_wrapper_t
                , bp::bases< base_i >
                , smart_pointers::smart_ptr_t<derived_wrapper_t> >( "derived_t" )
    // -----------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    // Pay attention on the class HeldType. It will allow us to create new classes
    // in Python, which derive from the derived_t class.
        .def( "get_value", &derived_t::get_value, &derived_wrapper_t::default_get_
↪value );

    // Now register all existing conversion:
    bp::implicitly_convertible<
                  smart_pointers::smart_ptr_t< derived_wrapper_t >
```

```
                , smart_pointers::smart_ptr_t< derived_t > >();
    bp::implicitly_convertible<
                smart_pointers::smart_ptr_t< derived_t >
                , smart_pointers::smart_ptr_t< base_i > >();
    bp::implicitly_convertible<
                  derived_ptr_t
                , smart_pointers::smart_ptr_t< derived_t > >();
    bp::register_ptr_to_python< derived_ptr_t >();

    bp::def( "const_ref_get_value", &::const_ref_get_value );
    bp::def( "ref_get_value", &::ref_get_value );
    bp::def( "val_get_value", &::val_get_value );
    bp::def( "create_derived", &::create_derived );
    bp::def( "create_base", &::create_base );


    bp::class_< numeric_t, smart_pointers::smart_ptr_t< numeric_t > >( "numeric_t" )
        .def_readwrite( "value", &numeric_t::value );

    bp::def( "create_numeric", &::create_numeric );
    bp::def( "get_numeric_value", &::get_numeric_value );

    // Work around for the public member variable, where type of the variable
    // is smart pointer problem
    bp::class_< shared_data::buffer_t >( "buffer_t" )
        .def_readwrite( "size", &shared_data::buffer_t::size );

    bp::register_ptr_to_python< smart_pointers::smart_ptr_t< shared_data::buffer_t > >
↪();

    bp::class_< shared_data::buffer_holder_t >( "buffer_holder_t" )
        .def( "get_data", &shared_data::buffer_holder_t::get_data )
        .def_readwrite( "data_naive", &shared_data::buffer_holder_t::data )
        // If you will try to access "data_naive" you will get
        // TypeError: No Python class registered for C++ class smart_pointers::smart_
↪ptr_t<shared_data::buffer_t>
        // Next lines of code contain work around
        .add_property( "data"
            , bp::make_getter( &shared_data::buffer_holder_t::data
                               , bp::return_value_policy<bp::copy_non_const_reference>
↪() )
            , bp::make_setter( &shared_data::buffer_holder_t::data ) );

}
```

Build script (SCons)

```
#scons build script
SharedLibrary( target=r'custom_sptr'
    , source=[ r'bindings.cpp' ]
    , LIBS=[ r"boost_python" ]
    , LIBPATH=[ r"/home/roman/boost_cvs/libs/python/build/bin-stage",r"" ]
    , CPPPATH=[ r"/home/roman/boost_cvs"
              , r"/usr/include/python2.4" ]
    , SHLIBPREFIX=''
    , SHLIBSUFFIX='.so'
```

```
)
```

Usage example/Tester

```
import unittest
import custom_sptr

class py_derived_t( custom_sptr.base_i ):
    def __init__( self ):
        custom_sptr.base_i.__init__( self )

    def get_value( self ):
        return 28

class py_double_derived_t( custom_sptr.derived_t ):
    def __init__( self ):
        custom_sptr.derived_t.__init__( self )

    def get_value( self ):
        return 0xDD

class tester_t( unittest.TestCase ):
    def __init__( self, *args ):
        unittest.TestCase.__init__( self, *args )

    def __test_ref( self, inst ):
        try:
            custom_sptr.ref_get_value( inst )
            self.fail( 'ArgumentError was not raised.' )
        except Exception, error:
            self.failUnless( error.__class__.__name__ == 'ArgumentError' )

    def __test_ref_fine( self, inst, val ):
        self.assertEqual( custom_sptr.ref_get_value( inst ), val )

    def __test_val( self, inst, val ):
        self.assertEqual( custom_sptr.val_get_value( inst ), val )

    def __test_const_ref( self, inst, val ):
        self.assertEqual( custom_sptr.const_ref_get_value( inst ), val )

    def __test_impl( self, inst, val ):
        self.__test_ref( inst )
        self.__test_val( inst, val )
        self.__test_const_ref( inst, val )

    def test_derived( self ):
        self.__test_impl( custom_sptr.derived_t(), 0xD )

    def test_py_derived( self ):
        self.__test_impl( py_derived_t(), 28 )

    def test_py_double_derived( self ):
        self.__test_impl( py_double_derived_t(), 0xDD )

    def test_created_derived( self ):
```

```
        self.__test_impl( custom_sptr.create_derived(), 0xD )

    def test_created_base( self ):
        inst = custom_sptr.create_base()
        val = 0xD
        self.__test_ref_fine( inst, val )
        self.__test_val( inst, val )
        self.__test_const_ref( inst, val )

    def test_mem_var_access( self ):
        holder = custom_sptr.buffer_holder_t()
        self.failUnless( holder.get_data().size == 0 )
        self.failUnless( holder.data.size == 0 )
        try:
            self.failUnless( holder.data_naive.size == 0 )
            self.fail("TypeError exception was not raised.")
        except TypeError:
            pass

    def test_numeric( self ):
        numeric = custom_sptr.create_numeric(21)
        self.failUnless( 21 == numeric.value )
        self.failUnless( 21 == custom_sptr.get_numeric_value(numeric) )
        numeric = custom_sptr.numeric_t()
        self.failUnless( 0 == numeric.value )
        self.failUnless( 0 == custom_sptr.get_numeric_value(numeric) )

def create_suite():
    suite = unittest.TestSuite()
    suite.addTest( unittest.makeSuite(tester_t))
    return suite

def run_suite():
    unittest.TextTestRunner(verbosity=2).run( create_suite() )

if __name__ == "__main__":
    run_suite()
```

All files contain comments, which describe what and why was done.

Download

smart_ptrs.zip

How to register `shared_ptr<const T>` conversion?

Solutions

There are two possible solutions to the problem. The first one is to fix Boost.Python library: *pointer_holder.hpp.patch* . The patch was contributed to the library (8-December-2006) and some day it will be committed to the CVS.

It is also possible to solve the problem, without changing Boost.Python library:

```
namespace boost{
```

```
    template<class T>
    inline T* get_pointer( boost::shared_ptr<const T> const& p ){
        return const_cast< T* >( p.get() );
    }

}

namespace boost{ namespace python{

    template<class T>
    struct pointee< boost::shared_ptr<T const> >{
        typedef T type;
    };

} } //boost::python

namespace utils{

    template< class T >
    register_shared_ptrs_to_python(){
        namespace bpl = boost::python;
        bpl::register_ptr_to_python< boost::shared_ptr< T > >();
        bpl::register_ptr_to_python< boost::shared_ptr< const T > >();
        bpl::implicitly_convertible< boost::shared_ptr< T >, boost::shared_
↪ptr< const T > >();
    }

}

BOOST_PYTHON_MODULE(...){
   class_< YourClass >( "YourClass" )
       ...;
   utils::register_shared_ptrs_to_python< YourClass >();
}
```

The second approach is a little bit “evil” because it redefines `get_pointer` function for all shared pointer class instantiations. So you should be careful.

Files

solution.cpp - C++ source file

```
#include "boost/python.hpp"
#include <string>

namespace boost{

    template<class T>
    inline T* get_pointer( boost::shared_ptr<const T> const& p ){
        return const_cast< T* >( p.get() );
    }

}

namespace boost{ namespace python{
```

```
    template<class T>
    struct pointee< boost::shared_ptr<T const> >{
        typedef T type;
    };

} } //boost::python

struct info_t{
    //class info_t records in what function it was created information
    info_t( const std::string& n )
    : text( n )
    {}

    std::string text;
};

typedef boost::shared_ptr< info_t > ptr_t;
typedef boost::shared_ptr< const info_t > const_ptr_t;

ptr_t create_ptr(){
    return ptr_t( new info_t( "ptr" ) );
}

const_ptr_t create_const_ptr(){
    return const_ptr_t( new info_t( "const ptr" ) );
}

std::string read_ptr( ptr_t x ){
    if( !x )
        return "";
    return x->text;
}

std::string read_const_ptr( const_ptr_t x ){
    if( !x )
        return "";
    return x->text;
}

namespace bpl = boost::python;

namespace utils{

    template< class T >
    void register_shared_ptrs_to_python(){
        //small helper function, which will register shared_ptr conversions
        bpl::register_ptr_to_python< boost::shared_ptr< T > >();
        bpl::register_ptr_to_python< boost::shared_ptr< const T > >();
        bpl::implicitly_convertible< boost::shared_ptr< T >, boost::shared_ptr< const␣
↪T > >();
    }

}

BOOST_PYTHON_MODULE( shared_ptr ){
    bpl::class_< info_t >( "info_t", bpl::init< std::string >() )
        .add_property( "text", &info_t::text );
    utils::register_shared_ptrs_to_python< info_t >();
```

```
    bpl::def( "create_ptr", &create_ptr );
    bpl::def( "create_const_ptr", &create_const_ptr );
    bpl::def( "read_ptr", &read_ptr );
    bpl::def( "read_const_ptr", &read_const_ptr );

}
```

Build script (SCons)

```
#scons build script
SharedLibrary( target=r'shared_ptr'
    , source=[ r'solution.cpp' ]
    , LIBS=[ r"boost_python" ]
    , LIBPATH=[ r"/home/roman/boost_cvs/bin",r"" ]
    , CPPPATH=[ r"/home/roman/boost_cvs"
              , r"/usr/include/python2.4" ]
    , SHLIBPREFIX=''
    , SHLIBSUFFIX='.so'
)
```

Usage example/tester

```
import unittest
import shared_ptr


class tester_t( unittest.TestCase ):
    def __init__( self, *args ):
        unittest.TestCase.__init__( self, *args )

    def test( self ):
        ptr = shared_ptr.create_ptr()
        self.failUnless( ptr.text == "ptr" )
        self.failUnless( shared_ptr.read_ptr( ptr ) == "ptr" )

        const_ptr = shared_ptr.create_const_ptr()
        self.failUnless( const_ptr.text == "const ptr" )
        self.failUnless( shared_ptr.read_const_ptr( const_ptr ) == "const ptr" )

        #testing conversion functionality
        self.failUnless( shared_ptr.read_const_ptr( ptr ) == "ptr" )

def create_suite():
    suite = unittest.TestSuite()
    suite.addTest( unittest.makeSuite(tester_t))
    return suite

def run_suite():
    unittest.TextTestRunner(verbosity=2).run( create_suite() )

if __name__ == "__main__":
    run_suite()
```

Boost.Python library patch

Download: pointer_holder.hpp.patch

```
*** pointer_holder.hpp.orig        2006-11-24 22:39:59.000000000 +0200
--- pointer_holder.hpp        2006-12-08 20:05:58.000000000 +0200
***************
*** 35,40 ****
--- 35,42 ----

  #  include <boost/detail/workaround.hpp>

+ #  include <boost/type_traits/remove_const.hpp>
+
  namespace boost { namespace python {

  template <class T> class wrapper;
***************
*** 122,146 ****
  template <class Pointer, class Value>
  void* pointer_holder<Pointer, Value>::holds(type_info dst_t, bool null_ptr_only)
  {
      if (dst_t == python::type_id<Pointer>()
          && !(null_ptr_only && get_pointer(this->m_p))
      )
          return &this->m_p;
!
!     Value* p = get_pointer(this->m_p);
      if (p == 0)
          return 0;

      if (void* wrapped = holds_wrapped(dst_t, p, p))
          return wrapped;

!     type_info src_t = python::type_id<Value>();
      return src_t == dst_t ? p : find_dynamic_type(p, src_t, dst_t);
  }

  template <class Pointer, class Value>
  void* pointer_holder_back_reference<Pointer, Value>::holds(type_info dst_t, bool␣
↪null_ptr_only)
  {
      if (dst_t == python::type_id<Pointer>()
          && !(null_ptr_only && get_pointer(this->m_p))
      )
--- 124,153 ----
  template <class Pointer, class Value>
  void* pointer_holder<Pointer, Value>::holds(type_info dst_t, bool null_ptr_only)
  {
+     typedef typename boost::remove_const< Value >::type NonConstValue;
+
      if (dst_t == python::type_id<Pointer>()
          && !(null_ptr_only && get_pointer(this->m_p))
      )
          return &this->m_p;
!
!     Value* tmp = get_pointer(this->m_p);
!     NonConstValue* p = const_cast<NonConstValue*>( tmp );
```

```
      if (p == 0)
          return 0;

      if (void* wrapped = holds_wrapped(dst_t, p, p))
          return wrapped;

!     type_info src_t = python::type_id<NonConstValue>();
      return src_t == dst_t ? p : find_dynamic_type(p, src_t, dst_t);
  }

  template <class Pointer, class Value>
  void* pointer_holder_back_reference<Pointer, Value>::holds(type_info dst_t, bool␣
↪null_ptr_only)
  {
+     typedef typename boost::remove_const< Value >::type NonConstValue;
+
      if (dst_t == python::type_id<Pointer>()
          && !(null_ptr_only && get_pointer(this->m_p))
      )
***************
*** 149,160 ****
      if (!get_pointer(this->m_p))
          return 0;

!     Value* p = get_pointer(m_p);

      if (dst_t == python::type_id<held_type>())
          return p;

!     type_info src_t = python::type_id<Value>();
      return src_t == dst_t ? p : find_dynamic_type(p, src_t, dst_t);
  }

--- 156,168 ----
      if (!get_pointer(this->m_p))
          return 0;

!     Value* tmp = get_pointer(this->m_p);
!     NonConstValue* p = const_cast<NonConstValue*>( tmp );

      if (dst_t == python::type_id<held_type>())
          return p;

!     type_info src_t = python::type_id<NonConstValue>();
      return src_t == dst_t ? p : find_dynamic_type(p, src_t, dst_t);
  }
```

Download

shared_ptr.zip

Automatic conversion between C++ and Python types

Introduction

Content

This example actually consist from 2 small, well documented examples.

The first one shows how to handle conversion between tuples: boost::tuples::tuple and Python tuple.

The second one shows how to add an automatic conversion from Python tuple to some registered class. The class registration allows you to use its functionality and enjoy from automatic conversion.

Files

Boost.Python to/from Python tuple conversion

```
// Copyright 2004-2007 Roman Yakovenko.
// Distributed under the Boost Software License, Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#ifndef TUPLES_HPP_16_JAN_2007
#define TUPLES_HPP_16_JAN_2007

#include "boost/python.hpp"
#include "boost/tuple/tuple.hpp"
#include "boost/python/object.hpp" //len function
#include <boost/mpl/int.hpp>
#include <boost/mpl/next.hpp>

/**
 * Converts boost::tuples::tuple<...> to\from Python tuple
 *
 * The conversion is done "on-the-fly", you should only register the conversion
 * with your tuple classes.
 * For example:
 *
 *  typedef boost::tuples::tuple< int, double, std::string > triplet;
 *  boost::python::register_tuple< triplet >();
 *
 * That's all. After this point conversion to\from next types will be handled
 * by Boost.Python library:
 *
 *   triplet
 *   triplet& ( return type only )
 *   const triplet
 *   const triplet&
 *
 * Implementation description.
 *  The conversion uses Boost.Python custom r-value converters. r-value converters
 * is very powerful and undocumented feature of the library. The only documentation
 * we have is http://boost.org/libs/python/doc/v2/faq.html#custom_string .
 *
 *  The conversion consists from two parts: "to" and "from".
 *
```

```
 *  "To" conversion
 * The "to" part is pretty easy and well documented ( http://docs.python.org/api/api.
↪html ).
 * You should use Python C API to create an instance of a class and than you
 * initialize the relevant members of the instance.
 *
 *  "From" conversion
 * Lets start from analyzing one of the use case Boost.Python library have to
 * deal with:
 *
 *   void do_smth( const triplet& arg ){...}
 *
 * In order to allow calling this function from Python, the library should keep
 * parameter "arg" alive until the function returns. In other words, the library
 * should provide instances life-time management. The provided interface is not
 * ideal and could be improved. You have to implement two functions:
 *
 *  void* convertible( PyObject* obj )
 *    Checks whether the "obj" could be converted to an instance of the desired
 *    class. If true, the function should return "obj", otherwise NULL
 *
 *  void construct( PyObject* obj, converter::rvalue_from_python_stage1_data* data)
 *    Constructs the instance of the desired class. This function will be called
 *    if and only if "convertible" function returned true. The first argument
 *    is Python object, which was passed as parameter to "convertible" function.
 *    The second object is some kind of memory allocator for one object. Basically
 *    it keeps a memory chunk. You will use the memory for object allocation.
 *
 *    For some unclear for me reason, the library implements "C style Inheritance"
 *    ( http://www.embedded.com/97/fe29712.htm ). So, in order to create new
 *    object in the storage you have to cast to the "right" class:
 *
 *      typedef converter::rvalue_from_python_storage<your_type_t> storage_t;
 *      storage_t* the_storage = reinterpret_cast<storage_t*>( data );
 *      void* memory_chunk = the_storage->storage.bytes;
 *
 *    "memory_chunk" points to the memory, where the instance will be allocated.
 *
 *    In order to create object at specific location, you should use placement new
 *    operator:
 *
 *      your_type_t* instance = new (memory_chunk) your_type_t();
 *
 *    Now, you can continue to initialize the instance.
 *
 *      instance->set_xyz = read xyz from obj
 *
 *    If "your_type_t" constructor requires some arguments, "read" the Python
 *    object before you call the constructor:
 *
 *      xyz_type xyz = read xyz from obj
 *      your_type_t* instance = new (memory_chunk) your_type_t(xyz);
 *
 *  Hint:
 * In most case you don't really need\have to work with C Python API. Let
 * Boost.Python library to do some work for you!
 *
 **/
```

```
namespace boost{ namespace python{

namespace details{

//Small helper function, introduced to allow short syntax for index incrementing
template< int index>
typename mpl::next< mpl::int_< index > >::type increment_index(){
    typedef typename mpl::next< mpl::int_< index > >::type next_index_type;
    return next_index_type();
}

}

template< class TTuple >
struct to_py_tuple{

    typedef mpl::int_< tuples::length< TTuple >::value > length_type;

    static PyObject* convert(const TTuple& c_tuple){
        list values;
        //add all c_tuple items to "values" list
        convert_impl( c_tuple, values, mpl::int_< 0 >(), length_type() );
        //create Python tuple from the list
        return incref( python::tuple( values ).ptr() );
    }

private:

    template< int index, int length >
    static void
    convert_impl( const TTuple &c_tuple, list& values, mpl::int_< index >, mpl::int_<␣
↪length > ) {
        values.append( c_tuple.template get< index >() );
        convert_impl( c_tuple, values, details::increment_index<index>(), length_
↪type() );
    }

    template< int length >
    static void
    convert_impl( const TTuple&, list& values, mpl::int_< length >, mpl::int_< length␣
↪>)
    {}

};


template< class TTuple>
struct from_py_sequence{

    typedef TTuple tuple_type;

    typedef mpl::int_< tuples::length< TTuple >::value > length_type;

    static void*
    convertible(PyObject* py_obj){

        if( !PySequence_Check( py_obj ) ){
```

```
            return 0;
        }

        if( !PyObject_HasAttrString( py_obj, "__len__" ) ){
            return 0;
        }

        python::object py_sequence( handle<>( borrowed( py_obj ) ) );

        if( tuples::length< TTuple >::value != len( py_sequence ) ){
            return 0;
        }

        if( convertible_impl( py_sequence, mpl::int_< 0 >(), length_type() ) ){
            return py_obj;
        }
        else{
            return 0;
        }
    }

    static void
    construct( PyObject* py_obj, converter::rvalue_from_python_stage1_data* data){
        typedef converter::rvalue_from_python_storage<TTuple> storage_t;
        storage_t* the_storage = reinterpret_cast<storage_t*>( data );
        void* memory_chunk = the_storage->storage.bytes;
        TTuple* c_tuple = new (memory_chunk) TTuple();
        data->convertible = memory_chunk;

        python::object py_sequence( handle<>( borrowed( py_obj ) ) );
        construct_impl( py_sequence, *c_tuple, mpl::int_< 0 >(), length_type() );
    }

    static TTuple to_c_tuple( PyObject* py_obj ){
        if( !convertible( py_obj ) ){
            throw std::runtime_error( "Unable to construct boost::tuples::tuple from␣
↪Python object!" );
        }
        TTuple c_tuple;
        python::object py_sequence( handle<>( borrowed( py_obj ) ) );
        construct_impl( py_sequence, c_tuple, mpl::int_< 0 >(), length_type() );
        return c_tuple;
    }

private:

    template< int index, int length >
    static bool
    convertible_impl( const python::object& py_sequence, mpl::int_< index >, mpl::int_
↪< length > ){

        typedef typename tuples::element< index, TTuple>::type element_type;

        object element = py_sequence[index];
        extract<element_type> type_checker( element );
        if( !type_checker.check() ){
            return false;
        }
```

```
        else{
            return convertible_impl( py_sequence, details::increment_index<index>(),␣
↪length_type() );
        }
    }

    template< int length >
    static bool
    convertible_impl( const python::object& py_sequence, mpl::int_< length >,␣
↪mpl::int_< length > ){
        return true;
    }

    template< int index, int length >
    static void
    construct_impl( const python::object& py_sequence, TTuple& c_tuple, mpl::int_<␣
↪index >, mpl::int_< length > ){

        typedef typename tuples::element< index, TTuple>::type element_type;

        object element = py_sequence[index];
        c_tuple.template get< index >() = extract<element_type>( element );

        construct_impl( py_sequence, c_tuple, details::increment_index<index>(),␣
↪length_type() );
    }

    template< int length >
    static void
    construct_impl( const python::object& py_sequence, TTuple& c_tuple, mpl::int_<␣
↪length >, mpl::int_< length > )
    {}

};

template< class TTuple>
void register_tuple(){

    to_python_converter< TTuple, to_py_tuple<TTuple> >();

    converter::registry::push_back( &from_py_sequence<TTuple>::convertible
                                  , &from_py_sequence<TTuple>::construct
                                  , type_id<TTuple>() );
};

} } //boost::python

#endif//TUPLES_HPP_16_JAN_2007
```

Boost.Python to/from Python tuple conversion tester

```
#include "boost/python.hpp"
#include "boost/tuple/tuple_comparison.hpp"
#include "tuples.hpp"

/**
```

```
 * Content:
 *  * few testers for  boost::tuples::tuple<...> to\from Python tuple conversion
 *    functionality
 *
 *  * example of custom r-value converter for a registered class
 *
 *
 **/

typedef boost::tuple< int, int, int > triplet_type;

triplet_type triplet_ret_val_000() {
    return triplet_type(0,0,0);
}

triplet_type triplet_ret_val_101() {
    return triplet_type(1,0,1);
}

triplet_type& triplet_ret_ref_010(){
    static triplet_type pt( 0,1,0 );
    return pt;
}

triplet_type* triplet_ret_ptr_110(){
    static triplet_type pt( 1,1,0 );
    return &pt;
}

bool test_triplet_val_000( triplet_type pt ){
    return pt == triplet_type( 0,0,0 );
}

bool test_triplet_cref_010( const triplet_type& pt ){
    return pt == triplet_type( 0,1,0 );
}

bool test_triplet_ref_110( triplet_type& pt ){
    return pt == triplet_type( 1,1,0 );
}

bool test_triplet_ptr_101( triplet_type* pt ){
    return pt && *pt == triplet_type( 1,0,1 );
}


namespace bpl = boost::python;

BOOST_PYTHON_MODULE( tuples ){

    bpl::register_tuple< triplet_type >();

    bpl::def("triplet_ret_val_000", &::triplet_ret_val_000);
    bpl::def("triplet_ret_val_101", &::triplet_ret_val_101);
    bpl::def("triplet_ret_ref_010"
             , &::triplet_ret_ref_010
             , bpl::return_value_policy<bpl:: copy_non_const_reference>() );
    bpl::def( "triplet_ret_ptr_110"
```

```
              , &::triplet_ret_ptr_110
              , bpl::return_value_policy<bpl::return_by_value>() );
    bpl::def("test_triplet_val_000", &::test_triplet_val_000);
    bpl::def("test_triplet_cref_010", &::test_triplet_cref_010);
    bpl::def("test_triplet_ref_110", &::test_triplet_ref_110);
    bpl::def("test_triplet_ptr_101", &::test_triplet_ptr_101);

}
```

Custom r-value converter registration

```
#include "boost/python.hpp"
#include "boost/python/object.hpp"  //len function
#include "boost/python/ssize_t.hpp" //ssize_t type definition
#include "boost/python/detail/none.hpp"
#include "tuples.hpp"

/**
 * Custom r-value converter example.
 *
 * Use-case description. I and few other developers work on Python bindings for
 * Ogre (http://ogre3d.org). The engine defines ColourValue class. This class
 * describes colour using 4 components: red, green, blue and transparency. The
 * class is used through the whole engine. One of the first features users ask
 * is to add an ability to pass a tuple, instead of the "ColourValue" instance.
 * This feature would allow them to write less code:
 *
 *   x.do_smth( (1,2,3,4) )
 *
 * instead of
 *
 *   x.do_smth( ogre.ColourValue( 1,2,3,4 ) )
 *
 * That's not all. They also wanted to be able to use ColourValue functionality.
 *
 * Solution.
 *
 * Fortunately, Boost.Python library provides enough functionality to implement
 * users requirements - r-value converters.
 *
 * R-Value converters allow one to register custom conallows one torom Python type to
 * C++ type. The conversion will be handled by Boost.Python library automatically
 * "on-the-fly".
 *
 * The example introduces "colour_t" class and few testers.
 *
 **/


struct colour_t{
    explicit colour_t( float red_=0.0, float green_=0.0, float blue_=0.0 )
    : red( red_ ), green( green_ ), blue( blue_ )
    {}

    bool operator==( const colour_t& other ) const{
        return red == other.red && green == other.green && blue == other.blue;
```

```
    }

    float red, green, blue;
};

struct desktop_t{
    bool is_same_colour( const colour_t& colour ) const{
        return colour == background;
    }
    colour_t background;
};

namespace bpl = boost::python;

struct pytuple2colour{

    typedef boost::tuples::tuple< float, float, float> colour_tuple_type;

    typedef bpl::from_py_sequence< colour_tuple_type > converter_type;

    static void* convertible(PyObject* obj){
        return converter_type::convertible( obj );
    }

    static void
    construct( PyObject* obj, bpl::converter::rvalue_from_python_stage1_data* data){
        typedef bpl::converter::rvalue_from_python_storage<colour_t> colour_storage_t;
        colour_storage_t* the_storage = reinterpret_cast<colour_storage_t*>( data );
        void* memory_chunk = the_storage->storage.bytes;

        float red(0.0), green(0.0), blue(0.0);
        boost::tuples::tie(red, green, blue) = converter_type::to_c_tuple( obj );

        colour_t* colour = new (memory_chunk) colour_t(red, green, blue);
        data->convertible = memory_chunk;
    }
};

void register_pytuple2colour(){
    bpl::converter::registry::push_back(  &pytuple2colour::convertible
                                        , &pytuple2colour::construct
                                        , bpl::type_id<colour_t>() );
}



bool test_val_010( colour_t colour ){
    return colour == colour_t( 0, 1, 0);
}

bool test_cref_000( const colour_t& colour ){
    return colour == colour_t( 0, 0, 0);
}

bool test_ref_111( colour_t& colour ){
    return colour == colour_t( 1, 1, 1);
}
```

```
bool test_ptr_101( colour_t* colour ){
    return colour && *colour == colour_t( 1, 0, 1);
}

bool test_cptr_110( const colour_t* colour ){
    return colour && *colour == colour_t( 1, 1, 0);
}

BOOST_PYTHON_MODULE( custom_rvalue ){
    bpl::class_< colour_t >( "colour_t" )
        .def( bpl::init< bpl::optional< float, float, float > >(
              ( bpl::arg("red_")=0.0, bpl::arg("green_")=0.0, bpl::arg("blue_")=0.0 )␣
↪) )
        .def_readwrite( "red", &colour_t::red )
        .def_readwrite( "green", &colour_t::green )
        .def_readwrite( "blue", &colour_t::blue );
    register_pytuple2colour();

    bpl::class_< desktop_t >( "desktop_t" )
        //naive aproach that will not work - plain Python assignment
        //.def_readwrite( "background", &desktop_t::background )
        //You should use properties to force the conversion
        .add_property( "background"
                       , bpl::make_getter( &desktop_t::background )
                       , bpl::make_setter( &desktop_t::background ) )
        .def( "is_same_colour", &desktop_t::is_same_colour );

    bpl::def("test_val_010", &::test_val_010);
    bpl::def("test_cref_000", &::test_cref_000);
    bpl::def("test_ref_111", &::test_ref_111);
    bpl::def("test_ptr_101", &::test_ptr_101);
    bpl::def("test_cptr_110", &::test_cptr_110);
}
```

Build script (SCons)

```
#scons build script
SharedLibrary( target=r'tuples'
    , source=[ r'tuples_tester.cpp' ]
    , LIBS=[ r"boost_python" ]
    , LIBPATH=[ r"/home/roman/boost_cvs/bin",r"" ]
    , CPPPATH=[ r"/home/roman/boost_cvs"
              , r"/usr/include/python2.4" ]
    , SHLIBPREFIX=''
    , SHLIBSUFFIX='.so'
)

SharedLibrary( target=r'custom_rvalue'
    , source=[ r'custom_rvalue.cpp' ]
    , LIBS=[ r"boost_python" ]
    , LIBPATH=[ r"/home/roman/boost_cvs/bin",r"" ]
    , CPPPATH=[ r"/home/roman/boost_cvs"
              , r"/usr/include/python2.4" ]
    , SHLIBPREFIX=''
    , SHLIBSUFFIX='.so'
)
```

Usage example/tester

```
import unittest
import tuples
import custom_rvalue

class tuplesersion_tester_t( unittest.TestCase ):
    def __init__( self, *args ):
        unittest.TestCase.__init__( self, *args )

    def test_tuples( self ):
        self.failUnless( (0,0,0) == tuples.triplet_ret_val_000() )
        self.failUnless( (1,0,1) == tuples.triplet_ret_val_101() )
        self.failUnless( (0,1,0) == tuples.triplet_ret_ref_010() )
        self.failUnlessRaises( TypeError, tuples.triplet_ret_ptr_110 )
        self.failUnless( tuples.test_triplet_val_000( (0,0,0) ) )
        self.failUnless( tuples.test_triplet_cref_010( (0,1,0) ) )
        self.failUnless( tuples.test_triplet_val_000( [0,0,0] ) )
        self.failUnless( tuples.test_triplet_cref_010( [0,1,0] ) )
        self.failUnlessRaises( TypeError, tuples.test_triplet_ref_110, (1,1,0) )
        self.failUnlessRaises( TypeError, tuples.test_triplet_ptr_101, (1,0,1) )

    def test_from_sequence( self ):
        self.failUnless( custom_rvalue.test_val_010( (0,1,0) ) )
        self.failUnless( custom_rvalue.test_cref_000( (0,0,0) ) )
        self.failUnless( custom_rvalue.test_val_010( [0,1,0] ) )
        self.failUnless( custom_rvalue.test_cref_000( [0,0,0] ) )
        self.failUnlessRaises( Exception, custom_rvalue.test_ref_111, (1,1,1) )
        self.failUnlessRaises( Exception, custom_rvalue.test_ptr_101, (1,0,1) )
        self.failUnlessRaises( Exception, custom_rvalue.test_cptr_110, (1,1,0) )

    def test_from_class( self ):
        color = custom_rvalue.colour_t
        self.failUnless( custom_rvalue.test_val_010( color(0,1,0) ) )
        self.failUnless( custom_rvalue.test_cref_000( color(0,0,0) ) )
        self.failUnless( custom_rvalue.test_ref_111( color(1,1,1) ) )
        self.failUnless( custom_rvalue.test_ptr_101( color(1,0,1) ) )
        self.failUnless( custom_rvalue.test_cptr_110( color(1,1,0) ) )

    def cmp_colours( self, c1, c2 ):
        return c1.red == c2.red and c1.green == c2.green and c1.blue == c2.blue

    def test_from_class_property( self ):
        colour = custom_rvalue.colour_t
        desktop = custom_rvalue.desktop_t()
        self.failUnless( self.cmp_colours( desktop.background, colour() ) )
        desktop.background = (1,0,1)
        self.failUnless( self.cmp_colours( desktop.background, colour(1,0,1) ) )
        self.failUnless( desktop.is_same_colour( (1,0,1) ) )
        self.failUnless( desktop.is_same_colour( colour(1,0,1 ) ) )

def create_suite():
    suite = unittest.TestSuite()
    suite.addTest( unittest.makeSuite(tuplesersion_tester_t))
    return suite

def run_suite():
    unittest.TextTestRunner(verbosity=2).run( create_suite() )
```

```
if __name__ == "__main__":
    run_suite()
```

Download

automatic_conversion.zip

Custom exceptions

Introduction

Boost.Python has a limitation: it does not allow one to create class, which derives from the class defined in Python. In most use cases this should not bother you, except one - exceptions. The example will provide you with one of the possible solutions.

What's the problem?

It is all about module interface and user expectations. If you can translate all your exceptions to built-in ones, than you are fine. You don't have to read this guide, but Boost.Python exception translator documentation.

My use case was different. I was supposed to export the exception classes and make them play nice with the `try ... except` mechanism. I mean, users should be able to:

1. "except" all exceptions using `except Exception, err:` statement
2. "except" the exposed library defined exception classes

I thought about few possible solutions to the problem. My first attempt was to add a missing functionality to Boost.Python library. Well, I quickly found out that the task is not a trivial one.

The following solution, I thought about, was to expose the exception class as-is and to define new class in Python, which derives from it and the built-in `Exception`. I implemented it and when I run the code I've got `TypeError`: "Error when calling the metaclass bases multiple bases have instance lay-out conflict".

The only solution left was to use "aggregation with automatic delegation". I mean instead of deriving from the exception class, I will keep it as a member variable in a class defined in Python, which derives from the built-in `Exception` one. Every time user access an attribute, the class defined in Python will automatically redirect the request to the variable. This technique is explained much better here: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52295 .

Files

All files contain comments, which describe what and why was done.

exceptions.cpp - C++ source code

```
#include "boost/python.hpp"
#include <stdexcept>
#include <iostream>
```

```
/**
 * Content:
 *  * example, which explain how to create custom exception class, which gives
 *    expected behaviour to exceptions exposed using Boost.Python library.
 *
 *  The example also allows you to map your exception classes to the Python
 *  built-in ones.
 *
 **/

class application_error : public std::exception{
public:

    application_error()
    : std::exception(), m_msg()
    {}

    application_error( const std::string& msg )
    : std::exception(), m_msg( msg )
    {}

    application_error( const application_error& other )
    : std::exception(other), m_msg( other.m_msg )
    {}

    const std::string& message() const
    { return m_msg; }

    virtual ~application_error() throw(){}

    std::string application_name() const
    { return "my_exceptions module"; }

private:
    const std::string m_msg;
};

//small dummy function that will conditionaly throws exception
void check_preconditions( bool raise_error ){
    if( raise_error ){
        throw application_error( "xyz" );
    }
}

//test function for custom r-value converter
std::string get_application_name( const application_error& err ){
    return err.application_name();
}

namespace bpl = boost::python;

struct exception_translator{

    exception_translator(){

        bpl::register_exception_translator<application_error>(&exception_
↪translator::translate);
```

```
        //Register custom r-value converter
        //There are situations, where we have to pass the exception back to
        //C++ library. This will do the trick
        bpl::converter::registry::push_back( &exception_translator::convertible
                                           , &exception_translator::construct
                                           , bpl::type_id<application_error>() );
    }

    static void
    translate( const application_error& err ){
        bpl::object pimpl_err( err );
        bpl::object pyerr_class = pimpl_err.attr( "py_err_class" );
        bpl::object pyerr = pyerr_class( pimpl_err );
        PyErr_SetObject( pyerr_class.ptr(), bpl::incref( pyerr.ptr() ) );
    }

    //Sometimes, exceptions should be passed back to the library.
    static void*
    convertible(PyObject* py_obj){
        if( 1 != PyObject_IsInstance( py_obj, PyExc_Exception ) ){
            return 0;
        }

        if( !PyObject_HasAttrString( py_obj, "_pimpl" ) ){
            return 0;
        }

        bpl::object pyerr( bpl::handle<>( bpl::borrowed( py_obj ) ) );
        bpl::object pimpl = bpl::getattr( pyerr, "_pimpl" );
        bpl::extract<application_error> type_checker( pimpl );
        if( !type_checker.check() ){
            return 0;
        }
        return py_obj;
    }

    static void
    construct( PyObject* py_obj, bpl::converter::rvalue_from_python_stage1_data* data)
↪{
        typedef bpl::converter::rvalue_from_python_storage<application_error> storage_
↪t;

        bpl::object pyerr( bpl::handle<>( bpl::borrowed( py_obj ) ) );
        bpl::object pimpl = bpl::getattr( pyerr, "_pimpl" );

        storage_t* the_storage = reinterpret_cast<storage_t*>( data );
        void* memory_chunk = the_storage->storage.bytes;
        application_error* cpp_err
            = new (memory_chunk) application_error( bpl::extract<application_error>
↪(pimpl) );

        data->convertible = memory_chunk;
    }

};

BOOST_PYTHON_MODULE( my_exceptions ){
```

```
    typedef bpl::return_value_policy< bpl::copy_const_reference > return_copy_const_
↪ref;
    bpl::class_< application_error >( "_application_error_" )
        .def( bpl::init<const std::string&>() )
        .def( bpl::init<const application_error&>() )
        .def( "application_name", &application_error::application_name)
        .def( "message", &application_error::message, return_copy_const_ref() )
        .def( "__str__", &application_error::message, return_copy_const_ref() );

    exception_translator();

    bpl::def( "check_preconditions", &::check_preconditions );
    bpl::def( "get_application_name", &::get_application_name );
}
```

Build script (SCons)

```
#scons build script
SharedLibrary( target=r'my_exceptions'
    , source=[ r'exceptions.cpp' ]
    , LIBS=[ r"boost_python" ]
    , LIBPATH=[ r"/home/roman/boost_cvs/libs/python/build/bin-stage" ]
    , CPPPATH=[ r"/home/roman/boost_cvs"
              , r"/usr/include/python2.4" ]
    , SHLIBPREFIX=''
    , SHLIBSUFFIX='.so'
)
```

Usage example/tester

```
import unittest
import my_exceptions

class application_error(Exception):
    def __init__( self, app_error ):
        Exception.__init__( self )
        self._pimpl = app_error

    def __str__( self ):
        return self._pimpl.message()

    def __getattribute__(self, attr):
        my_pimpl = super(application_error, self).__getattribute__("_pimpl")
        try:
            return getattr(my_pimpl, attr)
        except AttributeError:
            return super(application_error,self).__getattribute__(attr)

my_exceptions.application_error = application_error
my_exceptions._application_error_.py_err_class = application_error

class tester_t( unittest.TestCase ):
    def __init__( self, *args ):
        unittest.TestCase.__init__( self, *args )
```

```
    def test_function_call( self ):
        my_exceptions.check_preconditions( False )

    def test_concrete_error( self ):
        try:
            my_exceptions.check_preconditions( True )
        except my_exceptions.application_error, err:
            self.failUnless( str( err ) == "xyz" )

    def test_base_error( self ):
        try:
            my_exceptions.check_preconditions( True )
        except Exception, err:
            self.failUnless( str( err ) == "xyz" )

    def test_redirection( self ):
        try:
            my_exceptions.check_preconditions( True )
        except Exception, err:
            self.failUnless( err.application_name() == "my_exceptions module" )

    def test_converter( self ):
        try:
            my_exceptions.check_preconditions( True )
        except my_exceptions.application_error, err:
            app_name = my_exceptions.get_application_name( err )
            self.failUnless( "my_exceptions module" == app_name )


def create_suite():
    suite = unittest.TestSuite()
    suite.addTest( unittest.makeSuite(tester_t))
    return suite

def run_suite():
    unittest.TextTestRunner(verbosity=2).run( create_suite() )

if __name__ == "__main__":
    run_suite()
```

Download

exceptions.zip

4.7 Development history

4.7.1 Contributors

Thanks to all the people that have contributed patches, bug reports and suggestions:

- My wife - Yulia
- John Pallister

- Matthias Baas
- Allen Bierbaum
- Lakin Wecker
- Georgiy Dernovoy
- Gottfried Ganssauge
- Andy Miller
- Martin Preisler
- Meghana Haridev
- Julian Scheid
- Oliver Schweitzer
- Hernán Ordiales
- Bernd Fritzke
- Andrei Vermel
- Carsten(spom.spom)
- Pertti Kellomäki
- Benoît Leveau
- Nikolaus Rath
- Alan Birtles
- Minh-Tri Pham
- Aron Xu
- Mark Moll

4.7.2 Version 1.9

1. Update due ti changes in pygccxml 1.9.0.
2. “Indexing Suite V2” directory was removed. The code is located in “code_repository” one.
3. Ported to clang 3.8 compiler.
4. Many unittest were fixed.
5. GUI functionality was removed. Actually it stopped function years ago.
6. Support for GCC XML is removed. CastXML is the only supported generator.
7. Many documentation changes.
8. Adding “typedef” lookup functions to “module_builder_t” class.

4.7.3 Version 1.7

1. Update due to changes in pygccxml 1.8.0.
2. Performance improvements.
3. Small documentation fixes.

4.7.4 Version 1.6

1. Reorganize documentation, switch to readthedocs theme.
2. Misc. small fixes.

4.7.5 Version 1.1

1. Added support for Python 3.
2. Added support for pygccxml 1.7 and castxml.
3. Switched to setuptools instead of distutils.

4.7.6 Version 1.0.1 (unreleased)

1. The bug related to exposing free operators was fixed. Many thanks to Andrei Vermel.
2. Few bugs were fixed for 64Bit platform. Many thanks to Carsten.
3. `ctypes` backend was introduced - *Py++* is able to generate Python code, which uses `ctypes` package to call functions in DLLs or shared libraries.

   Massive refactoring, which preserve backward compatibility to previous releases, was done.
4. From now on, *Py++* will use Sphinx for all documentation.
5. *Indexing Suite V2* introduces few backward compatibility changes. The indexing suite became "headers only" library and doesn't requier Boost.Python library patching. See "*containers*" document for more information.
6. Support for *std::hash_map*<...> and *std::hash_set*<...> containers was added.
7. The bug related to transformed virtual function was fixed. Many thanks to Pertti Kellomäki.
8. Thanks to Benoît Leveau, the "Function Transformation" documentation is much better now.
9. The following transformers were added:
   - `inout_static_array`
   - `input_static_matrix`
   - `output_static_matrix`
   - `inout_static_matrix`

   Many thanks to Benoît Leveau.
10. Numerous bugs in "ctypes code generator" were fixed. Many thanks to Nikolaus Rath.
11. Thanks to Alan Birtles, for fixing a small issue on cygwin.
12. Thanks to Minh-Tri Pham, for reporting bug and providing patch for "from_address" transformer, on 64 bit platforms.
13. Thanks to Aron Xu, for pointing out that it is better to use "os.name", instead of "sys.platform" for platform specific logic
14. Thanks to Scott Sturdivant, for reporting the bug, related to bit fields code generation. The bug was fixed.

4.7.7 Version 1.0

1. The algorithm, which calculates what member functions should be redefined in derived class wrappers, was improved. Many thanks to Julian Scheid for the bug fix.

   The change explanation.

```
struct A{
    virtual void foo() {}
};

class B: public A{
};
```

   Previous version of *Py++* didn't generate wrapper for class `B`, even though `B` inherits `A`'s virtual function. Now if you have the following Python code:

```
class C(B):
  def __init__( self ):
      B.__init__(self)
  def foo(self):
      print "C.foo"
```

   then when `foo` is invoked on this instance on the C++ side of things, the Python code won't be executed as the wrapper was missing.

   Warning! There is a possibility that your generated code will not work! Keep reading.

   If you use "function transformation" functionality, than it is possible the generated code will **NOT** work. Consider the following example:

```
struct A{
    virtual void foo(int& i) {/*do smth*/}
};

class B: public A{
    virtual void foo(int& i) {/*do smth else*/}
};
```

   The *Py++* code:

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder_t( ... )
foo = mb.member_functions( 'foo' )
foo.add_transformation( FT.output(0) )
```

   The generated code, for class `B`, is:

```
namespace bp = boost::python;

struct B_wrapper : B, bp::wrapper< B > {
    virtual void foo( int & i ) const  { ... }

    static boost::python::tuple default_foo( ::B const & inst )
    { ... }

    virtual void foo( int & i ) const
```

```
    { ... }

    static boost::python::object default_foo( ::A const & inst )
    { ... }
};
...
bp::class_< B_wrapper, bp::bases< A > >( "B" )
    .def( "foo", (boost::python::tuple (*)( ::B const & ))( &B_wrapper::default_
↪foo ) )
    .def( "foo", (boost::python::object (*)( ::A const & ))( &B_wrapper::default_
↪foo ) );
```

As you can see, after applying the transformation both functions have same signature. Do you know what function will be called in some situation? I do - the wrong one :-(.

Unfortunately, there is no easy work around or some trick that you can use, which will not break the existing code. I see few solutions to the problem:

- change the alias of the functions

```
from pyplusplus import module_builder
from pyplusplus import function_transformers as FT

mb = module_builder_t( ... )
foo = mb.member_functions( '::A::foo' ).add_transformation( FT.output(0),␣
↪alias="foo_a" )
foo = mb.member_functions( '::B::foo' ).add_transformation( FT.output(0),␣
↪alias="foo_b" )
```

- use `inout` transformation - it preserves a function signature
- *Py++* can introduce a configuration, that will preserve the previous behaviour. I think this is a wrong way to go and doing the API changes is the 'right' longer term solution.

  If you **absolutely need** to preserve API backward compatible, contact me and I will introduce such configuration option.

Sorry for inconvenience.

2. Few bugs, related to Indexing Suite 2, were fixed. Many thanks to Oliver Schweitzer for reporting them.
3. New and highly experimental feature was introduced - *Boost.Python and ctypes integration*.
4. Support for *boost::python::make_constructor* functionality was added.
5. Support for unions and unnamed classes was added.
6. Doxygen documentation extractor was improved. Many thanks to Hernán Ordiales.
7. Py++ documentation was improved. Many thanks to Bernd Fritzke.

4.7.8 Version 0.9.5

1. Bug fixes:
   - Py++ will not expose free operators, if at least one of the classes, it works on, is not exposed. Many thanks to Meghana Haridev for reporting the bug.
2. Added ability to completely disable warnings reporting.
3. All logging is now done to `stderr` instead of `stdout`.

4. Generated code improvements:
   - `default_call_policies` is not generated
   - `return_internal_reference` call policies - default arguments are not generated
   - STD containers are generated without default arguments. For example instead of `std::vector< int, std::allocator< int > >`, in many cases *Py++* will generate `std::vector< int >`.
5. *create_with_signature* algorithm was improved. *Py++* will generate correct code in one more use case.
6. Added ability to exclude declarations from being exposed, if they will cause compilation to fail.
7. Starting from this version, *Py++* provides a complete solution for *multi-module development*.
8. Classes, which expose C arrays will be registered only once.
9. Starting from this version, *Py++* supports a code generation with different encodings.
10. There is a new strategy to split code into files. It is IDE friendly. Be sure to read *the updated documentation*.

4.7.9 Version 0.9.0

1. Bug fixes:
   - Declaration of virtual functions that have an exception specification with an empty throw was fixed. Now the exception specification is generated properly. Many thanks to Martin Preisler for reporting the bug.
2. Added exposing of copy constructor, `operator=` and `operator<<`.
   - `operator=` is exposed under "assign" name
   - `operator<<` is exposed under "__str__" name
3. Added new call policies:
   - *as_tuple*
   - *custom_call_policies*
   - *return_range*
4. Added an initial support for multi-module development. Now you can mark your declarations as `already_exposed` and *Py++* will do the rest. For more information read *multi-module development guide*.
5. *input_c_buffer* - new functions transformation, which allows one to pass a Python sequence to function, instead of pair of arguments: pointer to buffer and size.
6. Added ability to control generated "include" directives. Now you can ask *Py++* to include a header file, when it generates code for some declaration. For more information refers to inserting code guide.
7. Code generation improvements: system header files (Boost.Python or Py++ defined) will be included from the generated files only in case the generated code depends on them.
8. Performance improvements: Py++ runs 1.5 - 2 times faster, than the previous one.
9. Added ability to add code before overridden and default function calls. For more information refer to member function API documentation.
10. *Py++* will generate documentation for automatically constructed properties. For more information refer to *properties guide*.
11. Added iteration functionality to Boost.Python Indexing Suite V2 `std::map` and `std::multimap` containers.

4.7.10 Version 0.8.5

1. Added *Function Transformation* feature.
2. "Py++" introduces new functionality, which allows you to control messages and warnings: *how to disable warnings?*.
3. Added new algorithm, which controls the registration order of the functions. See *registration order document*
4. New "Py++" defined *return_pointee_value* call policy was introduced.
5. Support for opaque types was added. Read more about this feature here.
6. It is possible to configure "Py++" to generate faster (compilation time) code for indexing suite version 2. See API documentation.
7. The algorithm, which finds all class properties was improved. It provides user with a better way to control properties creation. A property that would hide another exposed declaration will not be registered\created.
8. Work around for "custom smart pointer as member variable" Boost.Python bug was introduced.
9. Bugs fixes and documentation improvement.

4.7.11 Version 0.8.2

1. Interface changes:
   - `module_builder.module_builder_t.build_code_creator` method: argument `create_casting_constructor` was removed and deprecation warning was introduced.
2. Performance improvements. In some cases you can get x10 performance boost. Many thanks to Allen Bierbaum! Saving and reusing results of different pygccxml algorithms and type traits functions achieved this.
3. Convenience API for registering exception translator was introduced.
4. *Py++* can generate code that uses `BOOST_PYTHON_FUNCTION_OVERLOADS` and `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` macros.
5. Treatment to previously generated and no more in-use files was added. By default *Py++* will delete these files, but of course you can redefine this behaviour.
6. Generated code changes:
   - `default_call_policies` should not be generated any more.
   - For functions that have `return_value_policy< return_opaque_pointer >` call policy, *Py++* will automatically generate `BOOST_PYTHON_OPAQUE_SPECIALIZED_TYPE_ID` macro. Thank you very much for Gottfried Ganssauge for this idea.
7. Support for Boost.Python properties was introduced. *Py++* implements small algorithm, that will automatically discover properties, base on naming conventions.
8. `decl_wrappers.class_t` has new function: `is_wrapper_needed`. This function explains why *Py++* creates class wrapper for exposed class.
9. Python type traits module was introduce. Today it contains only single function:
   - `is_immutable` - returns `True` if exposed type is Python immutable type

4.7.12 Version 0.8.1

1. Georgiy Dernovoy contributed a patch, which allows *Py++* GUI to save\load last used header file.
2. *Py++* improved a lot functionality related to providing feedback to user:
   - every package has its own logger
   - only important user messages are written to `stdout`
   - user messages are clear
3. Support for Boost.Python indexing suite version 2 was implemented.
4. Every code creator class took `parent` argument in `__init__` method. This argument was removed. `adopt_creator` and `remove_creator` will setunset reference to parent.
5. Generated code for member and free functions was changed. This changed was introduced to fix compilation errors on msvc 7.1 compiler.
6. *Py++* generates "stable" code. If header files were not changed, *Py++* will not change any file.
7. Support for huge classes was added. *Py++* is able to split registration code for the class to multiple cpp files.
8. User code could be added almost anywhere, without use of low level API.
9. Generated source files include only header files you passes as an argument to module builder.
10. Bug fixes.
11. Documentation was improved.

Project name changed

In this version the project has been renamed from "pyplusplus" to "Py++". There were few reasons to this:

1. I like "Py++" more then "pyplusplus".
2. "Py++" was the original name of the project: http://mail.python.org/pipermail/c++-sig/2005-July/009280.html
3. Users always changed the name of the projects. I saw at least 6 different names.

4.7.13 Version 0.8.0

1. *Py++* "user guide" functionality has been improved. Now *Py++* can answer few questions:
   - why this declaration could not be exported
   - why this function could not be overridden from Python
2. *Py++* can suggest an alias for exported classes.
3. Small redesign has been done - now it is much easier to understand and maintain code creators, which creates code for C++ functions.
4. Exception specification is taken into account, when *Py++* exports member functions.
5. Member variables, that are pointers exported correctly.
6. Added experimental support for `vector_indexing_suite`.
7. Bug fixes.

4.7.14 Version 0.7.0

Many thanks to *Matthias Baas* and *Allen Bierbaum*! They contributed so much to Py++, especially Matthias:

- New high-level API: *Py++* has simple and powerful API
- Documentation: Matthias and Allen added a lot of documentation strings
- Bug fixes and performance improvements

1. New GUI features:
   - It is possible now to see XML generated by CastXML.
   - It is possible to use GUI as wizard. It will help you to start with *Py++*, by generating *Py++* code.
2. **Attention - non backward compatible change**.

   `module_creator.creator_t.__init__` method has been changed. `decls` argument could be interpreted as
   - list of all declaration to be exported
   - list of top-level declarations. `creator_t` should export all declarations recursively.

   In order to clarify the use of `decls` argument new argument `recursive` has been added. By default new value of `recursive` is `False`.

   Guide for users/upgraders: if use are exporting all declaration without filtering, you should set `recursive` argument to `True`. If you use `pygccxml.declarations.filtering.*` functions, you have nothing to do.

   Sorry for the inconvenience :-(.
3. Better split of extension module to files. From now the following declarations will have dedicated file:
   - named enumerations, defined within namespace
   - unnamed enumerations and global variables
   - free functions

   This functionality will keep the number of instantiated templates within one file, `main.cpp`, to be very low. Also it is possible to implement solution, where `main.cpp` file does not contain templates instantiations at all.
4. Only constant casting operators could be used with `implicitly_convertible`. This bug has been fixed.
5. Bug exporting non copyable class has been fixed.
6. Small bug fix - from now file with identical content will not be overwritten.
7. Boost.Python `optional` is now supported and used when a constructor has a a default argument.
8. *Py++* now generates correct code for hierarchy of abstract classes:

```
struct abstract1{
    virtual void do_smth() = 0;
}

struct abstract2 : public abstract1{
    virtual void do_smth_else() = 0;
}

struct concrete : public abstract2{
    virtual void do_smth(){};
```

```
    virtual void do_smth_else(){};
}
```

9. Logging functionality has been added
10. New packages `module_builder`, `decl_wrappers` and `_logging_` has been added.
11. ...

http://boost.org/libs/python/doc/v2/init.html#optional-spec

4.7.15 Version 0.6.0

1. Code repository has been introduced. This repository contains classes and functions that will help users to export different C++ classes and declarations. Right now this repository contains two classes:
   - *array_1_t*
   - *const_array_1_t*

   Those classes helps to export static, single dimension arrays.
2. Code generation has been improved.
3. Code generation speed has been improved.
4. If you have Niall Douglas *void** patch, then you can enjoy from automatically set call policies.
5. Bit fields can be accessed from Python
6. Creating custom code creator example has been added.
7. Comparison to Pyste has been wrote.
8. Using this version it is possible to export most of TnFOX Python bindings.

4.7.16 Version 0.5.1

1. operator() is now supported.
2. Special casting operators are renamed(*__int__*, *__str__*, ...).
3. Few bug fixes

4.8 API

Py++ consists from few sub packages

4.8.1 pyplusplus.module_builder package

Overview

Modules

boost_python_builder

ctypes_builder

ctypes_decls_dependencies

module_builder

4.8.2 pyplusplus.decl_wrappers package

Overview

Modules

algorithm

calldef_wrapper

call_policies

class_wrapper

decl_wrapper

decl_wrapper_printer

doc_extractor

enumeration_wrapper

indexing_suite1

indexing_suite2

namespace_wrapper

properties

python_traits

scopedef_wrapper

typedef_wrapper

user_text

variable_wrapper

```
// file point.h
template< class T>
struct point_t{
    T x, y;
};

template <class T>
double distance( const point_t<T>& point ){
    return sqrt( point.x * point.x + point.y*point.y );
}

struct environment_t{
   ...
   template< class T>
   T get_value(const std::string& name);
   ...
};
```

Template instantiation

First of all you should understand, that you can not export template itself, but only its instantiations.

You can instantiate template class using operator `sizeof`:

```
sizeof( point_t<int> );
```

In order to instantiate a function you have to call it:

```
void instantiate(){
    double x = distance( point_t<t>() );

    environment_t env;
    std::string path = env.get_value< std::string >( "PATH" );
    int version = env.get_value< int >( "VERSION" );
}
```

You should put that code in some header file, parsed by CastXML.

"Dynamic" instantiation

If you have a template class, which should be instantiated with many types, you can create a small code generator, which will "instantiate the class". It is pretty easy to blend together the generated code and the existing one:

```
from module_builder import module_builder_t, create_text_fc

def generate_instantiations_string( ... ):
   ...

code = generate_instantiations_string( ... )

mb = module_builder_t( [ create_text_fc( code ), <<<other file names>>> ], ... )
...
```

Function `create_text_fc` allows you to extract declarations from the string, which contains valid C++ code. It creates temporal header file and compiles it.

Functions templated on return type

```
environment_t env;
std::string path = env.get_value< std::string >( "PATH" );
int version = env.get_value< int >( "VERSION" );
```

CastXML provides information for both instantiations:

- `get_value<int>`
- `get_value< std::string >`

But, in this case there is a catch: the name of both functions is "**get_value**". The only difference is "return type".

In this situation, *Py++* will generate code that contains errors. If your are lucky, it depends on the compiler you use, the generated code will not compile. Otherwise, you will discover the errors while testing the bindings.

Generated code:

```
bp::class_< environment_t >( "environment_t" )
    ...
    .def( "get_value"
          , (int ( ::environment_t::* )( ::std::string const & ) )( &::environment_
↪t::get_value ) )
    .def( "get_value"
          , (::std::string ( ::environment_t::* )( ::std::string const & ) )( &
↪::environment_t::get_value ) );
```

The correct code:

```
bp::class_< environment_t >( "environment_t" )
    .def( "get_value"
          , (int ( ::environment_t::* )( ::std::string const & ) )( &::environment_
↪t::get_value< int > ) )
          //-------------------------------------------------------------------------
↪^^^^^^^^^^^^^^^^
    .def( "get_value"
          , (::std::string ( ::environment_t::* )( ::std::string const & ) )( &
↪::environment_t::get_value< std::string > ) );
          //-------------------------------------------------------------------------
↪----------^^^^^^^^^^^^^^^^^^^^^^^^
```

The perfect one:

```
bp::class_< environment_t >( "environment_t" )
    ...
    .def( "get_value", &::environment_t::get_value< int > )
    .def( "get_value", &::environment_t::get_value< std::string > );
```

Work-around

Py++ contains a work-around to the problem:

```
mb = module_builder_t( ..., optimize_queries=False, ... )
environment = mb.class_( "environment_t" )
for f in environment.member_functions( "get_value" ):
    #set the function alias
    f.alias = f.name + "_" + f.return_type.decl_string
```

```
    #correct function name
    f.name = f.demangled_name
#you still want the queries to run fast
mb.run_query_optimizer()
```

Before you read the rest of the solution, you should understand what is "name mangling" means. If you don't, consider reading about it on Wikipedia .

The solution is pretty simple. CastXML reports mangled and demangled function names. The demangled function name contains "real" function name: `get_value< used type >`. You only have to instruct *Py++* to use it.

Py++ does not use by default demangled function name for mainly one reason. Demangled function name is a string that contains a lot of information. *Py++* implements a parser, which extracts the only relevant one. The parser implementation is a little bit complex and was not heavily tested. By "heavily" I mean that I tested it on a lot of crazy use cases and on a real project, but there is always some new use case out there. I am almost sure it will work for you. The problem, we deal with, is rare, so by default "demangled_name" feature is turned off.

By the way, almost the same problem exists for template classes. But, in the classes use case *Py++* uses demangled name by default.

Help wanted

I understand that the provided solutions are not perfect and that something better and simpler should be done. Unfortunatelly the priority of this task is low.

Allen Bierbaum has few suggestion that could improve *Py++*. He created a wiki page, that discuss possible solutions. Your contribution is welcome too!

4.9.2 How to register an exception translation?

Introduction

Boost.Python provides functionality to translate any C++ exception to a Python one. *Py++* provides a convenient API to do this.

By the way, be sure to take a look on "*troubleshooting guide - exceptions*". The guide will introduces a complete solution for handling exceptions within Python scripts.

Solution

Boost.Python exception translator documentation contains a complete explanation what should be done. I will use that example, to show how it could be done with *Py++*:

```
from pyplusplus import module_builder_t

mb = module_builder_t( ... )
my_exception = mb.class_( 'my_exception' )

translate_code = 'PyErr_SetString(PyExc_RuntimeError, exc.what();'
my_exception.exception_translation_code = translate_code
```

That's all, really. *Py++* will generate for you the `translate` function definition and than will register it.

I think this is a most popular use case - translate a C++ exception to a string and than to create an instance of Python built-in exception. That is exactly why *Py++* provides additional API:

```
mb = module_builder_t( ... )
my_exception = mb.class_( 'my_exception' )

my_exception.translate_exception_to_string( 'PyExc_RuntimeError', 'exc.what()')
```

The first argument of `translate_exception_to_string` method is exception type, The second one is a string - code that converts your exception to `const char*`.

As you see, it is really simple to add exception translation to your project.

One more point, in both pieces of code I used "`exc`" as the name of `my_exception` class instance. This is a predefined name. I am not going to change it without any good reason, any time soon :-).

4.9.3 Fatal error C1204: compiler limit: internal structure overflow

Fatal error C1204: compiler limit: internal structure overflow

If you get this error, than the generated file is too big. You will have to split it to few files. Well, not you but *Py++*, you will only have to tell it to do that.

If you are using `module_builder_t.write_module` method, consider to switch to `module_builder_t.split_module`.

If you are using `split_module`, but still the generated code for some class could not be compiled, because of the error, you can ask *Py++* to split the code generated for class to be split to few source files.

For more information, considre to read the *splitting generated code to files* document.

4.9.4 Absolute\relative paths

Absolute\relative paths

Consider the following layout:

```
boost/
  date_time/
    ptime.hpp
    time_duration.hpp
    date_time.hpp
```

`date_time.hpp` is the main header file, which should be parsed.

Py++ does not handle relative paths, as input, well. It tries, but there are uses cases it fails. In these cases it generates empty module - nothing is exposed:

```
mb = module_builder( [ 'date_time/date_time.hpp' ], ... )
mb.split_module( ... )
```

I recommend you to use absolute paths instead of relative ones:

```
import os
mb = module_builder( [ os.path.abspath('date_time/date_time.hpp') ], ... )
mb.split_module( ... )
```

and *Py++* will expose all declarations found in the `date_time.hpp` file and other files from the same directory.

4.9.5 Generated file name is too long

Generated file name is too long

There are use cases, when *Py++* generated file name is too long. In some cases the code generation process even fails because of this.

This is just a symptom of the problem. This happens when you expose template instantiated classes and you did not specify the class alias. *Py++* uses a class alias as a basis for the file name.

Let me explain.

```
template < class T>
struct holder{ ... };
```

As you know, a class name in Python has few constraints and *Py++* is aware of them. "holder< int >" is illegal class name, so *Py++* will generate another one - "holder_less_int_greater_". Pretty ugly and even long, but at least it is legal one.

It is pretty simple to change the alias of the class, or any other declaration:

```
from pyplusplus import module_builder

mb = module_builder_t( ... )
holder = mb.class_( 'holder< int >' )
holder.alias = 'IntHolder'
#the following line has same effect as the previous one:
holder.rename( 'IntHolder' )
```

Another solution to the problem, is to use different strategy to split the generated code to files. You can read more about splitting files *here*.

4.9.6 Best practices

Introduction

Py++ has reach interface and a lot of functionality. Sometimes reach interface helps, but sometimes it can confuse. This document will describe how effectively to use *Py++*.

Big projects

Definition

First of all, let me to define "big project". "Big project" is a project with few hundred of header files. *Py++* was born to create Python bindings for such projects. If you take a look *here* you will find few such projects.

Tips

- Create one header file, which will include all project header files.

  Doing it this way makes it so CastXML is only called once and it reduces the overhead that would occur if you pass CastXML all the files individually. Namely CastXML would have to run hundreds of times and each call would actually end up including quite a bit of common code anyway. This way takes a CastXML processing time from multiple hours with gigabytes of caches to a couple minutes with a reasonable cache size.

You can read more about different caches supported by pygccxml here. `module_builder_t.__init__` method takes reference to an instance of cache class or `None`:

```
from module_builder import *
mb = module_builder_t( ..., cache=file_cache_t( <<<path to project cache file>>>␣
↪), ... )
```

- Single header file, will also improve performance compiling the generated bindings.

  When *Py++* generated the bindings, you have a lot of .cpp files to compile. The project you are working on is big. I am sure it takes a lot of time to compile projects that depend on it. Generated code also depend on it, more over this code contains a lot of template instantiations. So it could take a great deal of time to compile it. Allen Bierbaum investigated this problem. He found out that most of the time is really spent processing all the headers, templates, macros from the project and from the boost library. So he come to conclusion, that in order to improve compilation speed, user should be able to control(to be able to generate) precompiled header file. He implemented an initial version of the functionality. After small discussion, we agreed on the following interface:

```
class module_builder_t( ... ):
    ...
    def split_module( self, directory_path, huge_classes=None, precompiled_
↪header=None ):
        ...
```

  `precompiled_header` argument could be `None` or string, that contains name of precompiled header file, which will be created in the directory. *Py++* will add to it header files from Boost.Python library and your header files.

  What is `huge_classes` argument for? `huge_classes` could be `None` or list of references to class declarations. It is there to provide a solution to this error. *Py++* will automatically split generated code for the huge classes to few files:

```
mb = module_builder_t( ... )
...
my_big_class = mb.class_( my_big_class )
mb.split_module( ..., huge_classes=[my_big_class], ... )
```

  - **Caveats**

    Consider the following file layout:

```
boost/
  date_time/
    ptime.hpp
    time_duration.hpp
    date_time.hpp //main header, which include all other header files
```

    Py++ currently does not handle relative paths as input very well, so it is recommended that you use "os.path.abspath()" to transform the header file to be processed into an absolute path:

```
#the following code will expose nothing
mb = module_builder( [ 'date_time/date_time.hpp' ], ... )

#while this one will work as expected
import os
mb = module_builder( [ os.path.abspath('date_time/date_time.hpp') ], ... )
```

- Keep the declaration tree small.

When parsing the header files to build the declaration tree, there will also be the occasional "junk" declaration inside the tree that is not relevant to the bindings you want to generate. These extra declarations come from header files that were included somewhere in the header files that you were actually parsing (e.g. if that library uses the STL or OpenGL or other system headers then the final declaration tree will contain those declarations, too). It can happen that the majority of declarations in your declaration tree are such "junk" declarations that are not required for generating your bindings and that just slow down the generation process (reading the declaration cache and doing queries will take longer).

To speed up your generation process you might want to consider making the declaration tree as small as possible and only store those declarations that somehow have an influence on the bindings. Ideally, this is done as early as possible and luckily CastXML provides an option that allows you to reduce the number of declarations that it will store in the output XML file. You can specify one or more declarations using the `-fxml-start` option and only those sub-tree starting at the specified declarations will be written. For example, if you specify the name of a particular class, only this class and all its members will get written. Or if your project already uses a dedicated namespace you can simply use this namespace as a starting point and all declarations stemming from system headers will be ignored (except for those declarations that are actually used within your library).

In the `pygccxml` package you can set the value for the `-fxml-start` option using the `start_with_declarations` attribute of the `pygccxml.parser.config_t` object that you are passing to the parser.

- Use *Py++* repository of generated files md5 sum.

  Py++ is able to store md5 sum of generated files in a file. Next time you will generate code, *Py++* will compare generated file content against the sum, instead of loading the content of the previously generated file from the disk and comparing against it.

```
mb = module_builder_t( ... )
...
my_big_class = mb.class_( my_big_class )
mb.split_module( ..., use_files_sum_repository=True )
```

  Py++ will generate file named "<your module name>.md5.sum" in the directory it will generate all the files.

  Enabling this functionality should give you 10-15% of performance boost.

  - **Caveats**

    If you changed manually some of the files - don't forget to delete the relevant line from "md5.sum" file. You can also delete the whole file. If the file is missing, *Py++* will use old plain method of comparing content of the files. It will not re-write "unchanged" files and you will not be forced to recompile the whole project.

4.9.7 Hints

Class template instantiation alias

Py++ has nice feature. If you define `typedef` for instantiated class template, than *Py++* will use it as a Python class name.

For example:

```
#include <vector>
typedef std::vector< int > numbers;
numbers generate_n(){
    ...
}
```

Py++ will use “numbers” as Python class name:

```
using boost::python;
class_< std::vector< int > >( "numbers" )
    ...
;
```

Py++ will pick up the alias, only in case the class has single “typedef”.

`pyplusplus::aliases` namespace

The previous approach is “implicit” - *Py++* does something behind the scene. Recently (version 0.8.6), another approach was introduced:

```
#include <vector>

namespace pyplusplus{ namespace aliases{
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    typedef std::vector< int > numbers;

} } //pyplusplus::aliases
```

The idea is that you create namespace with a special name - `pyplusplus::aliases` and *Py++* automatically picks the class aliases from it. In case you accidentally introduced two or more different aliases to the same class, it will pick the longest one and print a warning. Other advantages of the approach:

- you are not forced to learn new API
- you continue to use your favorite editor and familiar language

CHAPTER 5

Indices and tables

- genindex
- modindex
- search