Py++11 Documentation

Release 0.1.0

Rarios

Apr 28, 2017

Contents

CHAPTER 1

What is *Py++11*?

Definition: Py++11 is an object-oriented framework for creating a code generator for the pybind11 library and the ctypes package.

Py++11 uses a few different programming paradigms to help you expose C++ declarations in Python. This code generator will guide you through the whole process, raising warnings in the case you are doing something wrong, with a link to the explanation. And the most importantly: it will save you time. You will not have to update the code generator script every time the source code is changed.

CHAPTER 2

Code generation process

The code generation process consists of a few steps. The following paragraphs will tell you more about each step.

"read declarations"

Py++11 does not reinvent the wheel. It uses GCC C++ compiler to parse C++ source files. To be more precise, the tool chain looks like this:

1. Source code is passed to GCC-XML
2. GCC-XML passes it to GCC C++ compiler

3. GCC-XML generates an XML description of the C++ program from GCC's internal representation.

4. *Py++11* uses pygccxml package to read the GCC-XML generated file.

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

"build module"

Only very small and simple projects can 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 the question a little, what code should be written in order for me to get access from Python to that functionality?

Of course, *Py++11* cannot answer those question, but it provides as much help as it can.

How can *Py++11* help you with the first question? *Py++11* provides very a 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 of any type:

```
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 an '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 the code generator source code every time.

So far, so good. What about the second question? Well, by default *Py++11* generates code that will satisfy almost all developers. *Py++11* 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 the 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++11* allows you to do that.

Py++11 introduces new concepts: code creator and code creator tree. You can think about the code creator tree as some kind of AST. The only difference is that code creator trees provide more specific functionality. For example `include_t` code creator is responsible to create C++ `include` directive code. You have full control over the code creator tree, before it is written to disc. Here is an UML diagram of almost all code creators: class diagram.

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

"write code to files"

During this step *Py++11* reads the code creator tree and writes the code to disc. The code generation process result should not be different from the one a human would have created. For small projects, writing all code into single file is a good approach, however, for big ones the code should be splitted into multiple files. *Py++11* implements both strategies.

CHAPTER 3

Features list

- *Py++11* will supports almost all features found in pybind11 library. It currently generates *Boost.Python* code.
- You can develop extension modules simultaneously using *Py++11*, especially when they share code.
- *Py++11* generates code, which will help you:
  - understand compiler generated error messages
  - minimize project built time
- *Py++11* has a couple of 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++11* will check the "completeness" of the bindings. It will check for you that the exposed declarations don't have references to unexposed ones.
- *Py++11* provides enough functionality to extract source code documentation and write it as Python documentation string.
- *Py++11* provides a simple and powerful framework to create a wrapper for functions, which could not be exposed "as is" to Python.
- ...

CHAPTER 4

License

Boost Software License.

CHAPTER 5

Documentation contents

Tutorials

What is Py++?

Py++11 is an object-oriented framework for creating a code generator for the pybind11 library and the ctypes package.

Graphical interface

Py++ includes a graphical interface. Graphical interface is invoked with the `pyplusplus_gui` command, or with `pyplusplus_gui.pyw` from the `scripts` subdirectory, of the Python installation directory.

My advise to you - start with graphical interface, because:

- you don't have to learn new API
- few clicks with mouse and you have Boost.Python code for your file(s)
- it is very easy to evaluate Py++ using it
- you can check whether GCC-XML is able to compile your code or not
- you can use it as a guide to Boost.Python library
- it is able to generate Py++ code for you

Getting started

I suppose you decided to do some coding with Py++. Module builder tutorials will help you.

Advanced

To be written. I think I should cover here the usage of code creators and code creators tree. Meanwhile you can take a look on the content of `examples/custom_code_creator` directory. It contains example, which shows how to create your own code creator. To be more specific, it exposes `get*` and `set*` methods as a single property.

Users and quotes

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

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 developped 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 developping 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 scenegraph 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 .

Download & Install

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++

Subversion access

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

Download

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

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.

Dependencies

- pygccxml

Documentation

Help needed!

Py++ documentation is always under active development. It is not an easy task to create and maintain it. I will appreciate **any help**!

How can you help?

- Lets face it: today it is not possible to use Py++ without eventually looking into source code. Py++ uses Sphinx to generate documentation from source files. So, if you found some undocumented piece of code and you understand what it does, please write documentation string.
- You are reading documentation and my English cause you to scream? Please, fix those errors and send me new version of the document. I will integrate the changes.
- Do you think, that the documentation is not clear, I will be glad to improve it, just point me to the place.

Overview

Examples

Graphical interface

Py++ has nice, small and simple graphical interface. Consider to read *tutorials* for more information.

pyeasybmp

EasyBMP is a small cross-platform library that provide you functionality needed to work with Windows bitmap (BMP) image files. I took me only few minutes to create Python bindings for the library. Read more here.

boost libraries

Boost provides free peer-reviewed portable C++ source libraries. Using Py++ I created Python bindings for few libraries:

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

This is not "just another example". I went father and created new package: pyboost. This is fully working Python package, with almost all unit test from the libraries ported to Python. For more information please read pyboost package documentation.

Links

Wiki

Thanks to Allen Bierbaum Py++ now has wiki. We use it primary to discuss new features, which will be introduced in future versions.

Reading

- Building Hybrid Systems with Boost.Python
- A rationale for semantically enhanced library languages

  ".. A Semantically Enhanced Library Language (a SEL language or a SELL) is a dialect created by supersetting a language using a library and then subsetting the result using a tool that *understands* the syntax and semantics of both the underlying language and the library. ..."

  Py++ + Boost.Python is a SELL!
- Aspect oriented programming

  Py++ borrowed few ideas from this programming paradigm.

Help resources

- http://www.google.com :-)

  This site uses Google custom search engine, turned to provide better results when you search for materials related to Boost.Python library.

- http://boost.org/libs/python/doc/index.html - tutorials, FAQs, reference manuals
- Boost.Python wiki
- http://boost.cvs.sourceforge.net/boost/boost/libs/python/test/ - Boost.Python unit tests. They could be very, very helpful.
- http://pygccxml.svn.sourceforge.net/viewvc/pygccxml/pyplusplus_dev/unittests/ - Py++ unit tests. They could be even more helpful!

Mailing lists

- C++-sig - development of Python/C++ bindings
- Py++ mailing list

Libraries inspired by Boost.Python

- Luabind - interfacing C++ and Lua
- Pyd - interfacing C++ and D programming language
- Sq Plus - interfacing C++ and Squirrel

Projects inspired by Py++ :-)

- PyBindGen - a new project for producing Python extensions

Blogs

- http://www.shocksolution.com/math_tools/boost.python/index.html - this site contains few useful Boost.Python examples and tutorials.

Build systems

http://www.scons.org/wiki/GCCXMLBuilder - Joseph Lisee shows how to integrate Py++ scripts with Scons.

Compare Py++ to ...

Pyste

Pyste is the Boost.Python code generator, which is not under active development any more. Nevertheless, users request to compare Py++ and Pyste. You can read here the comparison.

SWIG & SIP

The document, that compares SIP, SWIG and Py++ is under construction. May be you are editing it right now, by evaluating these tools :-). I did not use SWIG and SIP, so I cannot provide you with fair comparison. I will let the open source project(s) "to talk":

- Python-OGRE:

  The impression of Lakin Wecker, after spending 30 hours working working with Py++: http://www.ogre3d.org/phpBB2addons/viewtopic.php?t=1478&sid=4d77585146aabbc54f4b31ec50874d86

  Python-OGRE project is reached the state, when it has all functionality provided by similar one - PyOgre. PyOgre is developed using SWIG. I suggest you to compare the amount of code, written by Python-Ogre developers and PyOgre ones:

  - PyOgre sources: http://svn.berlios.de/viewcvs/pyogre/trunk/pyogre/ogre/
  - Python-Ogre sources:

    http://python-ogre.svn.sourceforge.net/viewvc/python-ogre/trunk/python-ogre/code_generators/ogre/

    http://python-ogre.svn.sourceforge.net/viewvc/python-ogre/trunk/python-ogre/code_generators/common_utils/

    Pay attention: functionality defined in "common_utils" package is used by other scripts too.

Some other links, which compares Boost.Python, SWIG, SIP and other tools:

- Evaluation of Python/C++ interfacing packages
- Integrating Python, C and C++
- Python Wrapper Tools: A Performance Study

TODO

Description

This page is an official *Py++* "TODO" page.

For small features, the description of the feature and it's implementation will be written here. Big features will get their own page.

Boost.Python - lessons learned

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.

Development history

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

SVN Version

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 "../documentation/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.

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.mem_funs( '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.mem_funs( '::A::foo' ).add_transformation( FT.output(0), alias="foo_a
  ↪" )
  foo = mb.mem_funs( '::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.

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.

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 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.

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.

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

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.

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.

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 GCC-XML.
   - 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

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.

Version 0.5.1

1. operator() is now supported.
2. Special casting operators are renamed(*__int__*, *__str__*, ...).
3. Few bug fixes