pygccxml Documentation

Release 1.9.1

Insight Software Consortium

Oct 31, 2017

Contents

The purpose of *pygccxml* is to read a generated xml file and provide a simple framework to navigate C++ declarations, using Python classes.

Using *pygccxml* you can:

- Parse C++ source code
- Create a powerful code generator
- Generate UML diagrams
- Build code analyzers
- ...

CHAPTER 1

Installation

Installation instructions can be found here: *Installation*

CHAPTER 2

Examples

The *examples* are a good way to learn how to use *pygccxml*.

pygccxml provides a powerful API. If you want to know more about the provided API read the *query interface* document or the *API documentation*.

CHAPTER 3

Contributing

CHAPTER 4

License

Boost Software License.

CHAPTER 5

Test environment

pygccxml comes with comprehensive unit tests. They are executed on different operating systems, and with different versions of compilers. See the Travis builds for more details. *pygccxml* is tested under python 2.6, 2.7, 3.2, 3.3, 3.4, 3.5. All in all, *pygccxml* has more than 230 tests.

CHAPTER 6

Documentation contents

Download & Install

Prerequisite: CastXML

CastXML needs to be installed on your system.

1. If you are on linux or mac, your package manager may already provide a "castxml" package.
2. You can download pre-compiled binaries for Linux, for OS X and for Windows.
3. You can compile CastXML from source, either with the SuperBuild, or by following the full install instructions .

Installation of pygccxml

You can use pip to install pygccxml:

```
pip install pygccxml
```

To install from source, you can use the usual procedure:

```
python setup.py install
```

GCC-XML (Legacy)

These instructions are only here for historical reasons. GCC-XML was the tool used to generate the xml files before CastXML existed.

From version v1.8.0 on, pygccxml uses CastXML by default. The support for GCC-XML will finally be dropped in pygccxml v2.0.0.

There are few different ways to install GCC-XML on your system:

1. Most Linux system provide the "gccxml" package through their package manager.

2. See the instructions to install GCC-XML from source.

Examples

Setting up pygccxml and parsing c/c++ code

Parsing a c++ file

This example shows how to setup pygccxml to parse a c++ file, and how to access the declaration tree.

Let's consider the following c++ file (example.hpp):

```
namespace ns{
    int a = 1;
}
```

The following code will show you how to create a configuration for the xml generator (an external tool, either castxml or gccxml), and how to parse the c++ file:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find the location of the xml generator (castxml or gccxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

# Parse the c++ file
decls = parser.parse([filename], xml_generator_config)

# Get access to the global namespace
global_namespace = declarations.get_global_namespace(decls)

# Get access to the 'ns' namespace
ns = global_namespace.namespace("ns")
```

Parsing a string containing code

This example shows how to setup pygccxml to parse a string containing c++ code, and how to access the declaration tree. Often, pygccxml is used to parse files containing code, but there may be reasons to parse a string (for example for debugging purposes).

The following code will show you how to create a configuration for the xml generator (an external tool, either castxml or gccxml), and how to parse the string containing the c++ code:

```
from pygccxml import utils
from pygccxml import declarations
```

```
from pygccxml import parser

# Find the location of the xml generator (castxml or gccxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# Write a string containing some c++ code
code = """
    class MyClass {
        int a;
    };
"""

# Parse the code
decls = parser.parse_string(code, xml_generator_config)

# Get access to the global namespace
global_ns = declarations.get_global_namespace(decls)
```

First examples

Variables

This example shows how to find variables and find information about them.

Let's consider the following c++ file:

```
namespace ns{
    int a = 1;
    int b = 2;
    double c = 3.0;
}
```

The following code can be use to find the variable named "c" using different strategies, and to print information about it:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
```

```
global_namespace = declarations.get_global_namespace(decls)
ns = global_namespace.namespace("ns")

# The variables() method will return a list of variables.
# We know that the c variable is the third one in the list:
c = ns.variables()[2]
print("My name is: " + c.name)
print("My type is: " + str(c.decl_type))
print("My value is: " + c.value)

# Of course you can also loop over the list and look for the right name
for var in ns.variables():
    if var.name == "c":
        print("My name is: " + var.name)
        print("My type is: " + str(var.decl_type))
        print("My value is: " + var.value)

# One way to get a variable is to use the variable() method and
# a lambda function. This is the most flexible way as you can implement
# your own lambda function to filter out variables following your
# specific criteria.
c = ns.variable(lambda v: v.name == "c")
print("My name is: " + c.name)
print("My type is: " + str(c.decl_type))
print("My value is: " + c.value)
```

Searching for a declaration (using a loop)

This example shows how to search for a specific declaration using a loop on the declarations tree.

Let's consider the following c++ file (example.hpp):

```
namespace ns{
    int a = 1;
    int b = 2;
    double c = 3.0;

    double func2(double a) {
        double b = a + 2.0;
        return b;
    }
}
```

The following code will show you how to loop on the tree and find a declaration

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find the location of the xml generator (castxml or gccxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)
```

```
# The c++ file we want to parse
filename = "example.hpp"

# Parse the c++ file
decls = parser.parse([filename], xml_generator_config)

global_namespace = declarations.get_global_namespace(decls)

ns_namespace = global_namespace.namespace("ns")

int_type = declarations.cpptypes.int_t()
double_type = declarations.cpptypes.double_t()

for decl in ns_namespace.declarations:
    print(decl)

# This prints all the declarations in the namespace declaration tree:
# ns::a [variable]
# ns::b [variable]
# ns::c [variable]
# double ns::func2(double a) [free function]

# Let's search for specific declarations
for decl in ns_namespace.declarations:
    if decl.name == "b":
        print(decl)
    if isinstance(decl, declarations.free_function_t):
        print(decl)

# This prints:
# ns::b [variable]
# double ns::func2(double a) [free function]
```

Searching for a declaration (using matchers)

This example shows how to search for a specific declaration using different criteria.

Let's consider the following c++ file (example.hpp):

```
namespace ns{
    int a = 1;
    int b = 2;
    double c = 3.0;

    int func1(int a) {
        int b = a + 2;
        return b;
    }

    double func2(double a) {
        double b = a + 2.0;
        return b;
    }

     double func3(double a) {
        double b = a + 3.0;
        return b;
```

```
    }
}
```

The following code will show you how to search for functions and variables

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find the location of the xml generator (castxml or gccxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

# Parse the c++ file
decls = parser.parse([filename], xml_generator_config)

global_namespace = declarations.get_global_namespace(decls)

ns_namespace = global_namespace.namespace("ns")

int_type = declarations.cpptypes.int_t()
double_type = declarations.cpptypes.double_t()

# Search for the function by name
criteria = declarations.calldef_matcher(name="func1")
func1 = declarations.matcher.get_single(criteria, ns_namespace)

# Search for functions which return an int
criteria = declarations.calldef_matcher(return_type="int")
func2 = declarations.matcher.get_single(criteria, ns_namespace)

# Search for functions which return an int, using the cpptypes class
criteria = declarations.calldef_matcher(return_type=int_type)
func3 = declarations.matcher.get_single(criteria, ns_namespace)

print(func1)
print(func2)
print(func3)

# This prints:
# int ns::func1(int a) [free function]
# int ns::func1(int a) [free function]
# int ns::func1(int a) [free function]

# Search for functions which return a double. Two functions will be found
criteria = declarations.calldef_matcher(return_type=double_type)
func4 = declarations.matcher.find(criteria, ns_namespace)

print(len(func4))
print(func4[0])
print(func4[1])
```

```
# This prints:
# 2
# double ns::func2(double a) [free function]
# double ns::func3(double a) [free function]

# Finally, look for variables by name and by type
criteria = declarations.variable_matcher(name="a")
var_a1 = declarations.matcher.find(criteria, ns_namespace)

criteria = declarations.variable_matcher(decl_type=int_type)
var_a2 = declarations.matcher.find(criteria, ns_namespace)

print(var_a1[0])
print(var_a2[0])
print(var_a2[1])

# This prints:
# ns::a [variable]
# ns::a [variable]
# ns::b [variable]
```

Comparing two declarations

This example shows how two declarations can be compared.

Let's consider the following c++ file (example.hpp):

```
namespace ns{
    void func1(int a) {
        int b = a;
    }

    void func2(int a) {
        int b = a;
    }
}
```

The following code will show you how to search for two functions, using different ways. Both declarations are then compared.

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find the location of the xml generator (castxml or gccxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

# Parse the c++ file
```

```
decls = parser.parse([filename], xml_generator_config)

global_namespace = declarations.get_global_namespace(decls)

ns_namespace = global_namespace.namespace("ns")

# Search for the function called func1
criteria = declarations.calldef_matcher(name="func1")
func1a = declarations.matcher.get_single(criteria, ns_namespace)

# Search for the function called func2
criteria = declarations.calldef_matcher(name="func2")
func2a = declarations.matcher.get_single(criteria, ns_namespace)

# You can also write a loop on the declaration tree
func1b = None
for decl in ns_namespace.declarations:
    if decl.name == "func1":
        func1b = decl

# The declarations can be compared (prints (True, False))
print(func1a == func1b, func1a == func2a)
```

Functions and arguments

This example shows how to work with function arguments

Let's consider the following c++ file:

```
#include <iostream>
using namespace std;

namespace ns{
    int myFunction(int a, const std::string& x1) {
        return a + 1;
    }
}
```

The following code can be used to find the different arguments of a function and do some basic operations on them:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)
```

```
ns = global_namespace.namespace("ns")

# Use the free_functions method to find our function
func = ns.free_function(name="myFunction")

# There are two arguments:
print(len(func.arguments))

# We can loop over them and print some information:
for arg in func.arguments:
    print(
        arg.name,
        str(arg.decl_type),
        declarations.is_std_string(arg.decl_type),
        declarations.is_reference(arg.decl_type))
```

Nested types

This example shows how to work with types.

Let's consider the following c++ file:

```
namespace ns{
    const int a = 0;
    const volatile int *b = 0;
}
```

The following code allows you to extract information about the types of variables:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)
ns = global_namespace.namespace("ns")

a = ns.variables()[0]

print("My name is: " + a.name)
# > My name is: a

print("My type is: " + str(a.decl_type))
# > My type is: int const

# If we print real python type:
```

```
print("My type is : " + str(type(a.decl_type)))
# > My type is: <class 'pygccxml.declarations.cpptypes.const_t'>

# Types are nested in pygccxml. This means that you will get information
# about the first type only. You can access the "base" type by removing
# the const part:
print("My base type is: " + str(type(declarations.remove_const(a.decl_type))))
# > My base type is: <class 'pygccxml.declarations.cpptypes.int_t'>

# You use the is_const function to check for a type:
print("Is 'a' a const ?: " + str(declarations.is_const(a.decl_type)))
# > Is 'a' a const ?: True

# A more complex example with variable b:
b = ns.variables()[1]
print("My type is: " + str(type(b.decl_type)))
# > My type is: <class 'pygccxml.declarations.cpptypes.pointer_t'>
print("My type is: " + str(type(
    declarations.remove_const(
        declarations.remove_volatile(
            declarations.remove_pointer(b.decl_type))))))
# > My type is: <class 'pygccxml.declarations.cpptypes.int_t'>

# The declarations module contains much more methods allowing you to
# navigate the nested types list.
```

Explicit and implicit class declarations

Even if a class has no explicit constructor, `pygccxml` will provide a constructor declaration. This is due to `CastXML` and `GCC-XML` generating implicit constructors (for example copy constructors) in their XML output. The same thing holds for assignment operators and destructors.

To be able to discriminate between the different types of declarations, the `decl.is_artificial` attribute can be used.

Let's consider the following c++ file (example.hpp):

```
namespace ns{
    class Test {
       public:
         Test();  // This is the constructor
    };
}
```

In this example, the constructor is explicitly defined. The declaration tree will contain two constructors. The first one is the one we defined explicitly, and is not marked as artificial. The second one is the copy constructor, which was implicitly added, and is marked as artificial.

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
```

```
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)
ns = global_namespace.namespace("ns")

# We have just one declaration in ns, which is our Test class:
classTest = ns.declarations[0]
print(classTest.name, type(classTest))
# > 'Test', <class 'pygccxml.declarations.class_declaration.class_t'

# Loop over the two constructors:
for constructor in classTest.constructors():
    print(str(constructor), constructor.is_artificial)
# > ns::Test::Test() [constructor], False
# > ns::Test::Test(ns::Test const & arg0) [constructor], True
```

Compound types

A type is a compound_t type (in pygccxml) if it is one of the following: *volatile_t*, *restrict_t*, *const_t*, *pointer_t*, *reference_t*, *elaborated_t*, *array_t* or *member_variable_type_t*.

The exact c++ definition of compound types embraces more types, but for different reasons (mostly legacy), the definition in pygccxml is slightly different.

Let's consider the following c++ file:

```
int const c1 = 0;
const int c2 = 0;

volatile const int cv1 = 0;
const volatile int cv2 = 0;

const int * const cptr1 = 0;
```

The following code will show what to expect from compound types, how they are chained, and how their order is defined in pygccxml.

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"
```

```
decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)

c1 = global_namespace.variable("c1")
print(str(c1), type(c1))
# > 'c1 [variable]', <class 'pygccxml.declarations.variable.variable_t'>

print(str(c1.decl_type), type(c1.decl_type))
# > 'int const', <class 'pygccxml.declarations.cpptypes.const_t'>

base = declarations.remove_const(c1.decl_type)
print(str(base), type(base))
# > 'int', <class 'pygccxml.declarations.cpptypes.int_t'>

c2 = global_namespace.variable("c2")
print(str(c2.decl_type), type(c2.decl_type))
# > 'int const', <class 'pygccxml.declarations.cpptypes.const_t'>
# Even if the declaration was defined as 'const int', pygccxml will always
# output the const qualifier (and some other qualifiers) on the right hand
# side (by convention).

cv1 = global_namespace.variable("cv1")
print(str(cv1.decl_type), type(cv1.decl_type))
# > 'int const volatile', <class 'pygccxml.declarations.cpptypes.volatile_t'>

# Remove one level:
base = declarations.remove_volatile(cv1.decl_type)
print(str(base), type(base))
# > 'int const', <class 'pygccxml.declarations.cpptypes.const_t'>

# Remove the second level:
base = declarations.remove_const(base)
print(str(base), type(base))
# > 'int', <class 'pygccxml.declarations.cpptypes.int_t'>

# We can also directly do this in one step:
base = declarations.remove_cv(cv1.decl_type)
print(str(base), type(base))
# > 'int', <class 'pygccxml.declarations.cpptypes.int_t'>

# As for consts, the const and volatile are on the right hand side
# (by convention), and always in the same order
cv2 = global_namespace.variable("cv2")
print(str(cv2.decl_type), type(cv2.decl_type))
# > 'int const volatile', <class 'pygccxml.declarations.cpptypes.volatile_t'>

# And a last example with a pointer_t:
cptr1 = global_namespace.variable("cptr1")
print(str(cptr1.decl_type), type(cptr1.decl_type))
# > 'int const * const', <class 'pygccxml.declarations.cpptypes.const_t'>)

base = declarations.remove_const(cptr1.decl_type)
print(str(base), type(base))
# > 'int const *', <class 'pygccxml.declarations.cpptypes.pointer_t'>

base = declarations.remove_pointer(base)
print(str(base), type(base))
# > 'int const', <class 'pygccxml.declarations.cpptypes.const_t'>)
```

```
base = declarations.remove_const(base)
print(str(base), type(base))
# > 'int', <class 'pygccxml.declarations.cpptypes.int_t'>)
```

C++ Templates

pygccxml has minimal support for c++ templates, but there is some information that can be extracted from templated declarations.

Let's consider the following c++ file (example.hpp):

```
namespace ns {

struct B {
  struct D { bool d; };
};
struct D {};

template <typename T1, typename T2>
struct T {};

T<B::D, bool> function();

}
```

This example show how to extract template parameters from the template declaration.

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)
ns = global_namespace.namespace("ns")

class_t_decl = []
for d in ns.declarations:
    if isinstance(d, declarations.class_declaration_t):
        class_declaration_t = d
    if isinstance(d, declarations.class_t):
        class_t_decl.append(d)
    if isinstance(d, declarations.free_function_t):
        free_function_t_decl = d

print(class_t_decl[0])
```

```
# > ns::B [struct]

print(class_t_decl[1])
# > ns::D [struct]

print(class_declaration_t)
# > ns::T<ns::B::D, bool> [class declaration]

print(free_function_t_decl)
# > ns::T<ns::B::D, bool> ns::function() [free function]

print(declarations.templates.is_instantiation(class_declaration_t.name))
# > True

name, parameter_list = declarations.templates.split(class_declaration_t.name)
print(name, parameter_list)
# > 'T', ['ns::B::D', 'bool']
```

Advanced examples

Elaborated type specifiers

Elaborated type specifiers are one of these four possibilities: *class*, *struct*, *union* or *enum*.

In C++ they can often be skipped (but may be useful; see this interesting topic for example). In C code they are mandatory.

Let's consider the following c++ file:

```
class A {};

A a1;
class A a2;

void function(A arg1, class A arg2);
```

The following code will show how the elaborated type specifiers are treated in pygccxml. Please note that this feature is only available since recent versions of *CastXML* (Mar 1, 2017), and a special flag needs to be passed to pygccxml to make this work (castxml_epic_version=1).

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name,
    castxml_epic_version=1)

# The c++ file we want to parse
filename = this_module_dir_path + "/" + filename
```

```
decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)

a1 = global_namespace.variable("a1")
print(str(a1.decl_type), type(a1.decl_type))
# > 'A', <class 'pygccxml.declarations.cpptypes.declarated_t'>

print(declarations.is_elaborated(a1.decl_type))
# > False

a2 = global_namespace.variable("a2")
print(str(a2.decl_type), type(a2.decl_type))
# > 'class ::A', <class 'pygccxml.declarations.cpptypes.elaborated_t'>

print(declarations.is_elaborated(a2.decl_type))
# > True

base = declarations.remove_elaborated(a2.decl_type)
print(str(base), type(base))
# > 'A', <class 'pygccxml.declarations.cpptypes.declarated_t'>

# The same can be done with function arguments:
fun = global_namespace.free_function("function")
print(type(fun.arguments[0].decl_type), type(fun.arguments[1].decl_type))
# > <class 'pygccxml.declarations.cpptypes.declarated_t'>,
#   <class 'pygccxml.declarations.cpptypes.elaborated_t'>
```

Function pointers

This example shows how to work with function pointers.

Let's consider the following c++ file:

```
// A function pointer
void (*myFuncPointer)(int, double);
```

The following code allows you to extract information about the function pointer:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)

# The c++ file we want to parse
filename = "example.hpp"

decls = parser.parse([filename], xml_generator_config)
global_namespace = declarations.get_global_namespace(decls)

function_ptr = global_namespace.variables()[0]
```

```
# Print the name of the function pointer
print(function_ptr.name)
# > myFuncPointer

# Print the type of the declaration
print(function_ptr.decl_type)
# > void (*)( int,double )

# Print the real type of the declaration (it's just a pointer)
print(type(function_ptr.decl_type))
# > <class 'pygccxml.declarations.cpptypes.pointer_t'>

# Check if this is a function pointer
print(declarations.is_calldef_pointer(function_ptr.decl_type))
# > True

# Remove the pointer part, to access the function's type
f_type = declarations.remove_pointer(function_ptr.decl_type)

# Print the type
print(type(f_type))
# > <class 'pygccxml.declarations.cpptypes.free_function_type_t'>

# Print the return type and the arguments of the function
print(f_type.return_type)
# > void

# Print the return type and the arguments
print(str(f_type.arguments_types[0]), str(f_type.arguments_types[1]))
# > int, double
```

Caching

This example shows how to use caching. This can be useful for big projects where you don't want the c++ to be parsed again and again.

Let's consider the following c++ file:

```
namespace ns{
    int a = 1;
}
```

To enable caching, you can use the following code:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser

# Find out the c++ parser
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
xml_generator_config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name)
```

```
# The c++ file we want to parse
filename = "example.hpp"

file_config = parser.file_configuration_t(
    data=filename,
    content_type=parser.CONTENT_TYPE.CACHED_SOURCE_FILE)

project_reader = parser.project_reader_t(xml_generator_config)
decls = project_reader.read_files(
    [file_config],
    compilation_mode=parser.COMPILATION_MODE.FILE_BY_FILE)

global_namespace = declarations.get_global_namespace(decls)

value = global_namespace.namespace("ns")
print("My name is: " + value.name)
```

The first time you run this example, the c++ file will be read and a xml file will be generated:

INFO Creating xml file "example.hpp.xml" from source file "example.hpp" ... INFO Parsing xml file "example.hpp.xml" ... My name is: ns

The second time you run the example the xml file will not be regenerated:

INFO Parsing xml file "example.hpp.xml" ... My name is: ns

Of course the performance gain will be small for this example, but can be intersting for bigger projects.

Print all declarations

This example prints all declarations found in *example.hpp* file.

For every class, it prints it's base and derived classes.

The example consists from few files:

C++ header file

```
#ifndef example_hpp_12_10_2006
#define example_hpp_12_10_2006


namespace unittests{

struct test_results{

    enum status{ ok, fail, error };

    void update( const char* test_name, status result );
};

struct test_case{

    test_case( const char* test_case_name );

    virtual void set_up(){}
```

```
    virtual void tear_down(){}

    virtual void run() = 0;

private:
    const char* m_name;
};

class test_container;

struct test_suite : public test_case{

    test_suite( const char* name, const test_container& tests );

    void run();

    const test_results& get_results() const
    { return m_results; }

private:
    test_container* m_tests;
    test_results m_results;
};

}

#endif//example_hpp_12_10_2006
```

GCC-XML generated file

```
<?xml version="1.0"?>
<GCC_XML cvs_revision="1.127">
  <Namespace id="_1" name="::" members="_3 _4 _5 _6 _2 " mangled="_Z2::" demangled="::
↪"/>
  <Namespace id="_2" name="std" context="_1" members="" mangled="_Z3std" demangled=
↪"std"/>
  <Variable id="_3" name="_ZTIN9unittests10test_suiteE" type="_7c" context="_1"␣
↪location="f0:35" file="f0" line="35" extern="1" artificial="1"/>
  <Namespace id="_4" name="unittests" context="_1" members="_9 _10 _11 _12 " mangled=
↪"_Z9unittests" demangled="unittests"/>
  <Variable id="_5" name="_ZTIN9unittests9test_caseE" type="_13c" context="_1"␣
↪location="f0:19" file="f0" line="19" extern="1" artificial="1"/>
  <Namespace id="_6" name="__cxxabiv1" context="_1" members="" mangled="_Z10__cxxabiv1
↪" demangled="__cxxabiv1"/>
  <Struct id="_7" name="__si_class_type_info_pseudo" context="_1" mangled="27__si_
↪class_type_info_pseudo" demangled="__si_class_type_info_pseudo" location="f1:0"␣
↪file="f1" line="0" size="96" align="32" members=""/>
  <CvQualifiedType id="_7c" type="_7" const="1"/>
  <Struct id="_9" name="test_suite" context="_4" mangled="N9unittests10test_suiteE"␣
↪demangled="unittests::test_suite" location="f0:35" file="f0" line="35" artificial="1
↪" size="128" align="32" members="_15 _16 _17 _18 _19 _20 _21 _22 " bases="_12 ">
    <Base type="_12" access="public" virtual="0" offset="0"/>
  </Struct>
  <Struct id="_10" name="test_container" context="_4" incomplete="1" mangled=
↪"N9unittests14test_containerE" demangled="unittests::test_container" location="f0:33
↪" file="f0" line="33" artificial="1" align="8"/>
```

```
  <Struct id="_11" name="test_results" context="_4" mangled="N9unittests12test_
↪resultsE" demangled="unittests::test_results" location="f0:12" file="f0" line="12"␣
↪artificial="1" size="8" align="8" members="_23 _24 _25 _26 _27 _28 " bases=""/>
  <Struct id="_12" name="test_case" context="_4" abstract="1" mangled=
↪"N9unittests9test_caseE" demangled="unittests::test_case" location="f0:19" file="f0
↪" line="19" artificial="1" size="64" align="32" members="_29 _30 _31 _32 _33 _34 _
↪35 _36 " bases=""/>
  <Struct id="_13" name="__class_type_info_pseudo" context="_1" mangled="24__class_
↪type_info_pseudo" demangled="__class_type_info_pseudo" location="f1:0" file="f1"␣
↪line="0" size="64" align="32" members=""/>
  <CvQualifiedType id="_13c" type="_13" const="1"/>
  <Field id="_15" name="m_tests" type="_37" offset="64" context="_9" access="private"␣
↪location="f0:45" file="f0" line="45"/>
  <Field id="_16" name="m_results" type="_11" offset="96" context="_9" access="private
↪" location="f0:46" file="f0" line="46"/>
  <Destructor id="_17" name="test_suite" artificial="1" throw="" context="_9" access=
↪"public" mangled="_ZN9unittests10test_suiteD1Ev *INTERNAL* " demangled=
↪"unittests::test_suite::~test_suite()" location="f0:35" file="f0" line="35" endline=
↪"35" inline="1">
  </Destructor>
  <OperatorMethod id="_18" name="=" returns="_38" artificial="1" throw="" context="_9
↪" access="public" mangled="_ZN9unittests10test_suiteaSERKS0_" demangled=
↪"unittests::test_suite::operator=(unittests::test_suite const&amp;amp;)" location="f0:35
↪" file="f0" line="35" endline="35" inline="1">
    <Argument type="_39" location="f0:35" file="f0" line="35"/>
  </OperatorMethod>
  <Constructor id="_19" name="test_suite" artificial="1" throw="" context="_9" access=
↪"public" mangled="_ZN9unittests10test_suiteC1ERKS0_ *INTERNAL* " demangled=
↪"unittests::test_suite::test_suite(unittests::test_suite const&amp;amp;)" location=
↪"f0:35" file="f0" line="35" endline="35" inline="1">
    <Argument type="_39" location="f0:35" file="f0" line="35"/>
  </Constructor>
  <Constructor id="_20" name="test_suite" explicit="1" context="_9" access="public"␣
↪mangled="_ZN9unittests10test_suiteC1EPKcRKNS_14test_containerE *INTERNAL* "␣
↪demangled="unittests::test_suite::test_suite(char const*, unittests::test_container␣
↪const&amp;amp;)" location="f0:37" file="f0" line="37" extern="1">
    <Argument name="name" type="_40" location="f0:37" file="f0" line="37"/>
    <Argument name="tests" type="_41" location="f0:37" file="f0" line="37"/>
  </Constructor>
  <Method id="_21" name="run" returns="_42" virtual="1" overrides="_36 " context="_9"␣
↪access="public" mangled="_ZN9unittests10test_suite3runEv" demangled=
↪"unittests::test_suite::run()" location="f0:39" file="f0" line="39" extern="1"/>
  <Method id="_22" name="get_results" returns="_43" const="1" context="_9" access=
↪"public" mangled="_ZNK9unittests10test_suite11get_resultsEv" demangled=
↪"unittests::test_suite::get_results() const" location="f0:41" file="f0" line="41"␣
↪endline="42" inline="1"/>
  <Enumeration id="_23" name="status" context="_11" access="public" location="f0:14"␣
↪file="f0" line="14" artificial="1" size="32" align="32">
    <EnumValue name="ok" init="0"/>
    <EnumValue name="fail" init="1"/>
    <EnumValue name="error" init="2"/>
  </Enumeration>
  <Destructor id="_24" name="test_results" artificial="1" throw="" context="_11"␣
↪access="public" mangled="_ZN9unittests12test_resultsD1Ev *INTERNAL* " demangled=
↪"unittests::test_results::~test_results()" location="f0:12" file="f0" line="12"␣
↪endline="12" inline="1">
  </Destructor>
  <OperatorMethod id="_25" name="=" returns="_44" artificial="1" throw="" context="_11
↪" access="public" mangled="_ZN9unittests12test_resultsaSERKS0_" demangled=
↪"unittests::test_results::operator=(unittests::test_results const&amp;amp;)" location=
↪"f0:12" file="f0" line="12" endline="12" inline="1">
```

```
    <Argument type="_43" location="f0:12" file="f0" line="12"/>
  </OperatorMethod>
  <Constructor id="_26" name="test_results" artificial="1" throw="" context="_11"␣
↪access="public" mangled="_ZN9unittests12test_resultsC1ERKS0_ *INTERNAL* " demangled=
↪"unittests::test_results::test_results(unittests::test_results const&amp;amp;)"␣
↪location="f0:12" file="f0" line="12" inline="1">
    <Argument type="_43" location="f0:12" file="f0" line="12"/>
  </Constructor>
  <Constructor id="_27" name="test_results" artificial="1" throw="" context="_11"␣
↪access="public" mangled="_ZN9unittests12test_resultsC1Ev *INTERNAL* " demangled=
↪"unittests::test_results::test_results()" location="f0:12" file="f0" line="12"␣
↪inline="1"/>
  <Method id="_28" name="update" returns="_42" context="_11" access="public" mangled=
↪"_ZN9unittests12test_results6updateEPKcNS0_6statusE" demangled="unittests::test_
↪results::update(char const*, unittests::test_results::status)" location="f0:16"␣
↪file="f0" line="16" extern="1">
    <Argument name="test_name" type="_40" location="f0:16" file="f0" line="16"/>
    <Argument name="result" type="_23" location="f0:16" file="f0" line="16"/>
  </Method>
  <Field id="_29" name="m_name" type="_40" offset="32" context="_12" access="private"␣
↪location="f0:30" file="f0" line="30"/>
  <Destructor id="_30" name="test_case" artificial="1" throw="" context="_12" access=
↪"public" mangled="_ZN9unittests9test_caseD1Ev *INTERNAL* " demangled=
↪"unittests::test_case::~test_case()" location="f0:19" file="f0" line="19" endline=
↪"19" inline="1">
  </Destructor>
  <OperatorMethod id="_31" name="=" returns="_45" artificial="1" throw="" context="_12
↪" access="public" mangled="_ZN9unittests9test_caseaSERKS0_" demangled=
↪"unittests::test_case::operator=(unittests::test_case const&amp;amp;)" location="f0:19"␣
↪file="f0" line="19" endline="19" inline="1">
    <Argument type="_46" location="f0:19" file="f0" line="19"/>
  </OperatorMethod>
  <Constructor id="_32" name="test_case" artificial="1" throw="" context="_12" access=
↪"public" mangled="_ZN9unittests9test_caseC1ERKS0_ *INTERNAL* " demangled=
↪"unittests::test_case::test_case(unittests::test_case const&amp;amp;)" location="f0:19"␣
↪file="f0" line="19" endline="19" inline="1">
    <Argument type="_46" location="f0:19" file="f0" line="19"/>
  </Constructor>
  <Constructor id="_33" name="test_case" context="_12" access="public" mangled="_
↪ZN9unittests9test_caseC1EPKc *INTERNAL* " demangled="unittests::test_case::test_
↪case(char const*)" location="f0:21" file="f0" line="21" extern="1">
    <Argument name="test_case_name" type="_40" location="f0:21" file="f0" line="21"/>
  </Constructor>
  <Method id="_34" name="set_up" returns="_42" virtual="1" overrides="" context="_12"␣
↪access="public" mangled="_ZN9unittests9test_case6set_upEv" demangled=
↪"unittests::test_case::set_up()" location="f0:23" file="f0" line="23" inline="1"/>
  <Method id="_35" name="tear_down" returns="_42" virtual="1" overrides="" context="_
↪12" access="public" mangled="_ZN9unittests9test_case9tear_downEv" demangled=
↪"unittests::test_case::tear_down()" location="f0:25" file="f0" line="25" inline="1"/
↪>
  <Method id="_36" name="run" returns="_42" virtual="1" pure_virtual="1" overrides=""␣
↪context="_12" access="public" mangled="_ZN9unittests9test_case3runEv" demangled=
↪"unittests::test_case::run()" location="f0:27" file="f0" line="27" extern="1"/>
  <PointerType id="_37" type="_10" size="32" align="32"/>
  <ReferenceType id="_38" type="_9" size="32" align="32"/>
  <ReferenceType id="_39" type="_9c" size="32" align="32"/>
  <PointerType id="_40" type="_48c" size="32" align="32"/>
  <ReferenceType id="_41" type="_10c" size="32" align="32"/>
```

```
  <FundamentalType id="_42" name="void" align="8"/>
  <ReferenceType id="_43" type="_11c" size="32" align="32"/>
  <ReferenceType id="_44" type="_11" size="32" align="32"/>
  <ReferenceType id="_45" type="_12" size="32" align="32"/>
  <ReferenceType id="_46" type="_12c" size="32" align="32"/>
  <FundamentalType id="_48" name="char" size="8" align="8"/>
  <CvQualifiedType id="_48c" type="_48" const="1"/>
  <CvQualifiedType id="_11c" type="_11" const="1"/>
  <CvQualifiedType id="_12c" type="_12" const="1"/>
  <CvQualifiedType id="_10c" type="_10" const="1"/>
  <CvQualifiedType id="_9c" type="_9" const="1"/>
  <File id="f0" name="example.hpp"/>
  <File id="f1" name="&lt;built-in&gt;"/>
</GCC_XML>
```

Python API usage example

```
import os
import sys

# Find out the file location within the sources tree
this_module_dir_path = os.path.abspath(
    os.path.dirname(sys.modules[__name__].__file__))
# Add pygccxml package to Python path
sys.path.append(os.path.join(this_module_dir_path, '..', '..'))


from pygccxml import parser  # nopep8
from pygccxml import declarations  # nopep8
from pygccxml import utils  # nopep8

# Find out the xml generator (gccxml or castxml)
generator_path, generator_name = utils.find_xml_generator()

# Configure the xml generator
config = parser.xml_generator_configuration_t(
    xml_generator_path=generator_path,
    xml_generator=generator_name,
    compiler="gcc")

# Parsing source file
decls = parser.parse([this_module_dir_path + '/example.hpp'], config)
global_ns = declarations.get_global_namespace(decls)

# Get object that describes unittests namespace
unittests = global_ns.namespace('unittests')

print('"unittests" declarations: \n')
declarations.print_declarations(unittests)

# Print all base and derived class names
for class_ in unittests.classes():
    print('class "%s" hierarchy information:' % class_.name)
    print('\tbase classes   : ', repr([
        base.related_class.name for base in class_.bases]))
```

```
    print('\tderived classes: ', repr([
        derive.related_class.name for derive in class_.derived]))
    print('\n')

# Pygccxml has very powerfull query api:

# Select multiple declarations
run_functions = unittests.member_functions('run')
print('the namespace contains %d "run" member functions' % len(run_functions))
print('they are: ')
for f in run_functions:
    print('\t' + declarations.full_name(f))

# Select single declaration - all next statements will return same object
# vector< unittests::test_case* >

# You can select the class using "full" name
test_container_1 = global_ns.class_('::unittests::test_suite')
# You can define your own "match" criteria
test_container_2 = global_ns.class_(lambda decl: 'suite' in decl.name)

is_same_object = test_container_1 is test_container_2
print(
    "Does all test_container_* refer to the same object? " +
    str(is_same_object))
```

Output

```
>e:\Python26\pythonw.exe -u "example.py"
D:\dev\language-binding\sources\pygccxml_dev\docs\example\..\..
↪\pygccxml\parser\declarations_cache.py:8: DeprecationWarning: the md5 module is␣
↪deprecated; use hashlib instead
  import md5
INFO Parsing source file "example.hpp" ...
INFO gccxml cmd: ""D:\dev\language-binding\sources\gccxml_bin\v09\win32\bin\gccxml.exe
↪"  -I"."   "example.hpp" -fxml="e:\docume~1\romany\locals~1\temp\tmpewcrem.xml"  --
↪gccxml-compiler msvc71"
INFO GCCXML version - 0.9( 1.127 )
"unittests" declarations:

namespace_t: 'unittests'

    artificial: 'False'

    demangled: unittests

    mangled: _Z9unittests

    class_t: 'test_suite'

        location: [D:\dev\language-binding\sources\pygccxml_dev\docs\example\example.
↪hpp]:35

        artificial: '1'

        demangled: unittests::test_suite
```

```
        mangled: N9unittests10test_suiteE

        class type: 'struct'

        size: 16

        align: 4

        base classes:

            class: '::unittests::test_case'

                access type: 'public'

                virtual inheritance: 'False'

        public:

        destructor_t: '~test_suite'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:35

            artificial: '1'

            demangled: unittests::test_suite::~test_suite()

            mangled: _ZN9unittest

        member_operator_t: 'operator='

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:35

            artificial: '1'

            demangled: unittests::test_suite::operator=(unittests::test_suite const&)

            mangled: _ZN9unittests10test_suiteaSERKS0_

            is extern: False

            return type: ::unittests::test_suite &

            arguments type: ::unittests::test_suite const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

        constructor_t: 'test_suite'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:35
```

```
            artificial: '1'

            demangled: unittests::test_suite::test_suite(unittests::test_suite const&)

            mangled: _ZN9unittest

            is extern: False

            return type: None

            arguments type: ::unittests::test_suite const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: True

        constructor_t: 'test_suite'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:37

            artificial: 'False'

            demangled: unittests::test_suite::test_suite(char const*, unittests::test_
↪container const&)

            mangled: _ZN9unittest

            is extern: 1

            return type: None

            arguments type: char const * name, ::unittests::test_container const &␣
↪tests

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: False

        member_function_t: 'run'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:39

            artificial: 'False'
```

```
            demangled: unittests::test_suite::run()

            mangled: _ZN9unittests10test_suite3runEv

            is extern: 1

            return type: void

            arguments type:

            calling convention: __thiscall__

            virtual: virtual

            is const: False

            is static: False

        member_function_t: 'get_results'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:41

            artificial: 'False'

            demangled: unittests::test_suite::get_results() const

            mangled: _ZNK9unittests10test_suite11get_resultsEv

            is extern: False

            return type: ::unittests::test_results const &

            arguments type:

            calling convention: __thiscall__

            virtual: not virtual

            is const: 1

            is static: False

        protected:

        private:

        variable_t: 'm_tests'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:45

            artificial: 'False'

            type: ::unittests::test_container *

            value: None
```

```
            size: 4

            align: 4

            offset: 8


        variable_t: 'm_results'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:46

            artificial: 'False'

            type: ::unittests::test_results

            value: None

            size: 1

            align: 1

            offset: 12


    class_declaration_t: 'test_container'

        location: [D:\dev\language-binding\sources\pygccxml_dev\docs\example\example.
↪hpp]:33

        artificial: '1'

        demangled: unittests::test_container

        mangled: N9unittests14test_containerE

    class_t: 'test_results'

        location: [D:\dev\language-binding\sources\pygccxml_dev\docs\example\example.
↪hpp]:12

        artificial: '1'

        demangled: unittests::test_results

        mangled: N9unittests12test_resultsE

        class type: 'struct'

        size: 1

        align: 1

        public:

        enumeration_t: 'status'
```

```
            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:14

            artificial: '1'

            values:



                ok : 0

                fail : 1

                error : 2

        destructor_t: '~test_results'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:12

            artificial: '1'

            demangled: unittests::test_results::~test_results()

            mangled: _ZN9unittest

        member_operator_t: 'operator='

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:12

            artificial: '1'

            demangled: unittests::test_results::operator=(unittests::test_results␣
↪const&)

            mangled: _ZN9unittests12test_resultsaSERKS0_

            is extern: False

            return type: ::unittests::test_results &

            arguments type: ::unittests::test_results const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

        constructor_t: 'test_results'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:12
```

```
            artificial: '1'

            demangled: unittests::test_results::test_results(unittests::test_results␣
↪const&)

            mangled: _ZN9unittest

            is extern: False

            return type: None

            arguments type: ::unittests::test_results const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: True

        constructor_t: 'test_results'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:12

            artificial: '1'

            demangled: unittests::test_results::test_results()

            mangled: _ZN9unittest

            is extern: False

            return type: None

            arguments type:

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: False

        member_function_t: 'update'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:16

            artificial: 'False'

            demangled: unittests::test_results::update(char const*, unittests::test_
↪results::status)
```

```
            mangled: _ZN9unittests12test_results6updateEPKcNS0_6statusE

            is extern: 1

            return type: void

            arguments type: char const * test_name, ::unittests::test_results::status␣
↪result

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

        protected:

        private:

    class_t: 'test_case'

        location: [D:\dev\language-binding\sources\pygccxml_dev\docs\example\example.
↪hpp]:19

        artificial: '1'

        demangled: unittests::test_case

        mangled: N9unittests9test_caseE

        class type: 'struct'

        size: 8

        align: 4

        derived classes:

            class: '::unittests::test_suite'

                access type: 'public'

                virtual inheritance: 'False'

        public:

        destructor_t: '~test_case'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:19

            artificial: '1'

            demangled: unittests::test_case::~test_case()
```

```
            mangled: _ZN9unittest

        member_operator_t: 'operator='

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:19

            artificial: '1'

            demangled: unittests::test_case::operator=(unittests::test_case const&)

            mangled: _ZN9unittests9test_caseaSERKS0_

            is extern: False

            return type: ::unittests::test_case &

            arguments type: ::unittests::test_case const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

        constructor_t: 'test_case'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:19

            artificial: '1'

            demangled: unittests::test_case::test_case(unittests::test_case const&)

            mangled: _ZN9unittest

            is extern: False

            return type: None

            arguments type: ::unittests::test_case const & arg0

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: True

        constructor_t: 'test_case'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:21
```

```
            artificial: 'False'

            demangled: unittests::test_case::test_case(char const*)

            mangled: _ZN9unittest

            is extern: 1

            return type: None

            arguments type: char const * test_case_name

            calling convention: __thiscall__

            virtual: not virtual

            is const: False

            is static: False

            copy constructor: False

        member_function_t: 'set_up'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:23

            artificial: 'False'

            demangled: unittests::test_case::set_up()

            mangled: _ZN9unittests9test_case6set_upEv

            is extern: False

            return type: void

            arguments type:

            calling convention: __thiscall__

            virtual: virtual

            is const: False

            is static: False

        member_function_t: 'tear_down'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:25

            artificial: 'False'

            demangled: unittests::test_case::tear_down()

            mangled: _ZN9unittests9test_case9tear_downEv
```

```
            is extern: False

            return type: void

            arguments type:

            calling convention: __thiscall__

            virtual: virtual

            is const: False

            is static: False

        member_function_t: 'run'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:27

            artificial: 'False'

            demangled: unittests::test_case::run()

            mangled: _ZN9unittests9test_case3runEv

            is extern: 1

            return type: void

            arguments type:

            calling convention: __thiscall__

            virtual: pure virtual

            is const: False

            is static: False

        protected:

        private:

        variable_t: 'm_name'

            location: [D:\dev\language-binding\sources\pygccxml_
↪dev\docs\example\example.hpp]:30

            artificial: 'False'

            type: char const *

            value: None

            size: 4

            align: 4
```

```
            offset: 4


class "test_suite" hierarchy information:
        base classes    :  ['test_case']
        derived classes:  []


class "test_results" hierarchy information:
        base classes    :  []
        derived classes:  []


class "test_case" hierarchy information:
        base classes    :  []
        derived classes:  ['test_suite']


the namespace contains 2 "run" member functions
they are:
        ::unittests::test_suite::run
        ::unittests::test_case::run
Does all test_container_* refer to the same object?  True
>Exit code: 0
```

FAQ

GCCXML vs CastXML

GCCXML has been superseded by CastXML. It is highly recommended to use CastXML. GCCXML support will be removed from Pygccxml in version 2.0.

C++ and C code support

Pygccxml supports C++98, as CastXML and GCCXML only output declarations from the C++98 subset. Of course, newer versions of C++ can be parsed (the tests currently all pass with C++11 and C++14), but not all new features from these language definitions can be used.

C code support has been reported to work. As C is similar to C++, this makes sense. Some discrepancies may be present.

Still, parsing C code is not officially supported by pygccxml, as it falls out of scope of this project. Of course, if some volunteer wants to work on this, submissions would be accepted.

Function and method bodies

Pygccxml does not allow to fetch declarations defined in function or method bodies. For example the following a variable will not appear in the declarations tree:

```
int f() {
    int a = 3;
```

```
    return a;
}
```

Neither `GCCXML` or `CastXML` currently support this feature. `CastXML` could probably be extended for this later, as `pygccxml`.

Performance

pygccxml is being regularly optimised for performance, but it still may be slow in certain cases.

Before all, it is highly recommended to benchmark your application if performance is important to you. There are multiple tools out there for benchmarking python applications. We currently are using the following two command lines / tools:

```
python -m cProfile -o profile_data.pyprof script_to_profile.py
pyprof2calltree -i profile_data.pyprof -k
```

Of course optimising pygccxml alone will not help in all cases. The bottlenecks can also be in the code calling pygccxml, to make sure to benchmark the whole process. Any help on the performance side is also welcome.

Some things you may try (in order of priority):

1. 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 and gccxml provide an option that allows you to reduce the number of declarations that need to be parsed.

   You can specify one or more declarations using the `-fxml-start` (gccxml) or `-castxml-start` (castxml) options when running the xml generator. For example, if you specify the name of a particular class, only this class and all its members will get written. Ideally, your project should already use a dedicated namespace, which you can then use as a starting point. 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 these flags by using the `start_with_declarations` attribute of the `pygccxml.parser.config_t` object that you are passing to the parser.

2. You can pass the following flag to the *read_files* method:

   ```
   compilation_mode=pygccxml.parser.COMPILATION_MODE.ALL_AT_ONCE
   ```

3. If you want to cache the declarations tree, there is a caching mechanism provided by pygccxml. You will find an example of this mechanism in the examples section.

Flags

castxml_epic_version

The `` `castxml_epic_version` `` can be set to 1 to benefit from new castxml and pygccxml features. To be able to use this, you will need the latest castxml version.

Currently this adds the support for elaborated type specifiers.

__va_list_tag and other hidden declarations (f1)

When parsing with CastXML, the XML tree can contain declarations named `__va_list_tag`. If the compiler is llvm 3.9, `__NSConstantString_tag` and `__NSConstantString` declarations may also be present.

These declarations are internal declarations, coming from the std c++ library headers you include, and are often not needed. They are for example polluting the declarations tree when running pyplusplus.

By default, pygccxml will ignore these declarations. To still read these declarations from the xml file, a config flag can be set (`config.flags = ["f1"]`), or a flag can be passed as argument the config setup (`flags=["f1"]`).

__thiscall__ in attributes (f2)

Attributes defined as `` `__thiscall__` `` are now ignored (tested with VS 2013). The `` `__thiscall__` `` in some attributes will be removed too. If you still want to have access to these attributes, you can use the `config.flags = ["f2"]` option.

API

pygccxml package

Python CastXML or GCC-XML front end.

This package provides functionality to extract and inspect declarations from C/C++ header files. This is accomplished by invoking an external tool like CastXML or GCC-XML, which parses a header file and dumps the declarations as a XML file. This XML file is then read by pygccxml and the contents are made available as appropriate Python objects.

To parse a set of C/C++ header files you use the *parse* function in the :mod:parser sub package which returns a tree that contains all declarations found in the header files. The root of the tree represents the main namespace *::* and the children nodes represent the namespace contents such as other namespaces, classes, functions, etc. Each node in the tree is an object of a type derived from the `declaration_t` class. An inner node is always either a namespace `declarations.namespace_t` or a class `declarations.class_t`, which are both derived from `declarations.scopedef_t` class. Everything else (free functions, member functions, enumerations, variables, etc.) are always a leaf. You will find all those declaration classes in the :mod:declarations sub-package.

Subpackages

pygccxml.declarations package

Contains classes that describe different C++ declarations

access_type_matcher
see `access_type_matcher_t` for documentation

alias of `access_type_matcher_t`

and_matcher
see `and_matcher_t` for documentation

alias of `and_matcher_t`

calldef_matcher
see `calldef_matcher_t` for documentation

alias of `calldef_matcher_t`

custom_matcher
see `custom_matcher_t` for documentation

alias of `custom_matcher_t`

declaration_matcher
see declaration_matcher_t for documentation

alias of declaration_matcher_t

namespace_matcher
see namespace_matcher_t for documentation

alias of namespace_matcher_t

not_matcher
see not_matcher_t for documentation

alias of not_matcher_t

operator_matcher
see operator_matcher_t for documentation

alias of operator_matcher_t

or_matcher
see or_matcher_t for documentation

alias of or_matcher_t

regex_matcher
see regex_matcher_t for documentation

alias of regex_matcher_t

variable_matcher
see variable_matcher_t for documentation

alias of variable_matcher_t

virtuality_type_matcher
see virtuality_type_matcher_t for documentation

alias of virtuality_type_matcher_t

Submodules

pygccxml.declarations.algorithm module

Define few unrelated algorithms that work on declarations.

apply_visitor(*visitor*, *decl_inst*)
Applies a visitor on declaration instance.

Parameters **visitor** (type_visitor_t or decl_visitor_t) – instance

class match_declaration_t(*decl_type=None*, *name=None*, *fullname=None*, *parent=None*)
Bases: object

Helper class for different search algorithms.

This class will help developer to match declaration by:

- **declaration type, for example class_t or** operator_t.
- declaration name
- declaration full name
- reference to parent declaration

does_match_exist(*inst*)
Returns True if inst does match one of specified criteria.

Parameters **inst** (declaration_t) – declaration instance

Return type bool

pygccxml.declarations.algorithms_cache module

Defines classes that will keep results of different calculations.

class declaration_algs_cache_t
Bases: object

access_type

cmp_data
Data used for comparison between declarations.

container_element_type

container_key_type

container_traits

declaration_path

demangled_name

disable()

enable()

enabled

full_name

full_partial_name

normalized_full_name_false

normalized_full_name_true

normalized_name

normalized_partial_name

partial_declaration_path

reset()

reset_access_type()

reset_name_based()

class type_algs_cache_t
Bases: object

decl_string

static disable()

static enable()

enabled = True

partial_decl_string

remove_alias

reset()

pygccxml.declarations.byte_info module

class byte_info
Bases: object

This class stores information about the byte size and byte align values from a declaration/type.

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

pygccxml.declarations.call_invocation module

Free function invocation parser

The parser is able to extract function name and list of arguments from a function invocation statement. For example, for the following code

```
do_something( x1, x2, x3 )
```

the parser will extract - function name - *do_something* - argument names - *[x1, x2, x3]*

args(*declaration_string*)
Returns list of function arguments

Return type [str]

find_args(*text*, *start=None*)
Finds arguments within function invocation.

Return type [arguments] or :data:NOT_FOUND if arguments could not be found.

is_call_invocation(*declaration_string*)
Returns True if *declaration_string* is a function invocation.

Parameters **declaration_string** (*str*) – string that should be checked for pattern.

Return type bool

join(*name_*, *args_*, *arg_separator=None*)
Returns name(argument_1, argument_2, ..., argument_n).

name(*declaration_string*)
Returns the name of a function.

Return type str

split(*declaration_string*)
Returns (name, [arguments])

split_recursive(*declaration_string*)
Returns [(name, [arguments])].

pygccxml.declarations.calldef module

defines classes, that describes "callable" declarations

This modules contains definition for next C++ declarations:

- **operator**
  - member
  - free
- **function**
  - member
  - free
- constructor
- destructor

class argument_t(*name=''*, *decl_type=None*, *default_value=None*, *attributes=None*)
Bases: `object`

class, that describes argument of "callable" declaration

attributes
GCCXML attributes, set using __attribute__((gccxml("..."))) @type: str

clone(***keywd*)
constructs new argument_t instance

return argument_t(name=keywd.get('name', self.name), decl_type=keywd.get('decl_type', self.decl_type), default_value=keywd.get('default_value', self.default_value), attributes=keywd.get('attributes', self.attributes))

decl_type

default_value
Argument's default value or None. @type: str

ellipsis
bool, if True argument represents ellipsis ("...") in function definition

name
Argument name. @type: str

class calldef_t(*name=''*, *arguments=None*, *exceptions=None*, *return_type=None*, *has_extern=False*, *does_throw=True*, *mangled=None*)
Bases: *pygccxml.declarations.declaration.declaration_t*

base class for all "callable" declarations

argument_types
list of all argument types

arguments
The argument list. @type: list of *argument_t*

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

get_mangled_name()

guess_calling_convention()
This function should be overriden in the derived classes and return more-or-less successfull guess about calling convention

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of *calldef_t*

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

pygccxml.declarations.calldef_members module

class casting_operator_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef_members.member_calldef_t*, *pygccxml.declarations.calldef_members.operator_t*

describes casting operator declaration

OPERATOR_WORD_LEN = 8

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

symbol
operator's symbol. For example – operator+, symbol is equal to '+'

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class constructor_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef_members.member_calldef_t*

describes constructor declaration

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

explicit
True, if constructor has "explicit" keyword, False otherwise @type: bool

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class destructor_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef_members.member_calldef_t*

describes deconstructor declaration

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class member_calldef_t (*virtuality=None*, *has_const=None*, *has_static=None*, **args*, ***keywords*)
Bases: *pygccxml.declarations.calldef.calldef_t*

base class for "callable" declarations that defined within C++ class or struct

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class member_function_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef_members.member_calldef_t*

describes member function declaration

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class member_operator_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef_members.member_calldef_t*, *pygccxml.declarations.calldef_members.operator_t*

describes member operator declaration

OPERATOR_WORD_LEN = 8

access_type
Return the access type of the member (as defined by the string constants in the class :class:ACCESS_TYPES. @type: str

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()

has_const
describes, whether "callable" has const modifier or not

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

has_static
describes, whether "callable" has static modifier or not

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

symbol
operator's symbol. For example – operator+, symbol is equal to '+'

top_parent
Reference to top parent declaration.

@type: declaration_t

virtuality
Describes the "virtuality" of the member (as defined by the string constants in the class :class:VIRTUALITY_TYPES). @type: str

class operator_t(**args*, ***keywords*)

Bases: *pygccxml.declarations.calldef.calldef_t*

Base class for "operator" declarations.

Operators are constructs which behave like functions. Therefore, operator_t has calldef_t as parent class.

OPERATOR_WORD_LEN = 8

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

get_mangled_name()

guess_calling_convention()
This function should be overriden in the derived classes and return more-or-less successfull guess about calling convention

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

symbol
operator's symbol. For example – operator+, symbol is equal to '+'

top_parent
Reference to top parent declaration.

@type: declaration_t

pygccxml.declarations.calldef_types module

class CALLING_CONVENTION_TYPES
Bases: `object`

class that defines "calling convention" constants

CDECL = 'cdecl'

FASTCALL = 'fastcall'

STDCALL = 'stdcall'

SYSTEM_DEFAULT = '<<<system default>>>'

THISCALL = 'thiscall'

UNKNOWN = ''

all = ('', 'cdecl', 'stdcall', 'thiscall', 'fastcall', '<<<system default>>>')

static **extract**(*text*, *default=''*)
extracts calling convention from the text. If the calling convention could not be found, the "default"is used

pattern = <_sre.SRE_Pattern object at 0x32e9670>

FUNCTION_VIRTUALITY_TYPES
alias of *VIRTUALITY_TYPES*

class VIRTUALITY_TYPES
Bases: object

class that defines "virtuality" constants

ALL = ['not virtual', 'virtual', 'pure virtual']

NOT_VIRTUAL = 'not virtual'

PURE_VIRTUAL = 'pure virtual'

VIRTUAL = 'virtual'

pygccxml.declarations.class_declaration module

defines classes, that describes C++ classes

This modules contains definition for next C++ declarations:

- class definition
- class declaration
- small helper class for describing C++ class hierarchy

class ACCESS_TYPES
Bases: object

class that defines "access" constants

ALL = ['public', 'private', 'protected']

PRIVATE = 'private'

PROTECTED = 'protected'

PUBLIC = 'public'

class CLASS_TYPES
Bases: object

class that defines "class" type constants

ALL = ['class', 'struct', 'union']

CLASS = 'class'

STRUCT = 'struct'

UNION = 'union'

class class_declaration_t(*name=''*)
Bases: *pygccxml.declarations.declaration.declaration_t*

describes class declaration

aliases
List of `aliases` to this instance

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

get_mangled_name()

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

name
Declaration name @type: str

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

top_parent
Reference to top parent declaration.

@type: declaration_t

class class_t(*name=''*, *class_type='class'*, *is_abstract=False*)
Bases: *pygccxml.declarations.scopedef.scopedef_t*, *pygccxml.declarations.byte_info.byte_info*, *pygccxml.declarations.elaborated_info.elaborated_info*

describes class definition

ALLOW_EMPTY_MDECL_WRAPPER = False

RECURSIVE_DEFAULT = True

USE_DEMANGLED_AS_NAME = True

adopt_declaration(*decl*, *access*)
adds new declaration to the class

Parameters

- **decl** – reference to a `declaration_t`
- **access** (*:class:ACCESS_TYPES*) – member access type

aliases
List of `aliases` to this instance

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

bases
list of *base classes*

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calldef(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to "calldef" declaration, that is matched defined criteria

calldefs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of `calldef_t` declarations, that are matched defined criteria

casting_operator(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to casting operator declaration, that is matched defined criteria

casting_operators(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of casting operator declarations, that are matched defined criteria

class_(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to class declaration, that is matched defined criteria

class_type
describes class *type*

classes(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of class declarations, that are matched defined criteria

clear_optimizer()
Cleans query optimizer state

constructor(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to constructor declaration, that is matched defined criteria

constructors(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of constructor declarations, that are matched defined criteria

create_decl_string(*with_defaults=True*)

decl(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to declaration, that is matched defined criteria

decl_string
Declaration full name.

declaration_not_found_t

declarations
List of children declarations.

Returns List[declarations.declaration_t]

decls(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of declarations, that are matched defined criteria

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

derived
list of *derived classes*

elaborated_type_specifier
Elaborated specifier (can be – struct, union, class or enum).

Returns elaborated specifier

Return type str

enum(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the enumeration() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

enumeration(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to enumeration declaration, that is matched defined criteria

enumerations(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of enumeration declarations, that are matched defined criteria

enums(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the enumerations() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

find_out_member_access_type(*member*)
returns member access type

Parameters **member** (declaration_t) – member of the class

Return type :class:ACCESS_TYPES

get_mangled_name()

get_members(*access=None*)
returns list of members according to access type

If access equals to None, then returned list will contain all members. You should not modify the list content, otherwise different optimization data will stop work and may to give you wrong results.

Parameters **access** (*:class:ACCESS_TYPES*) – describes desired members

Return type [*members*]

i_depend_on_them(*recursive=True*)

init_optimizer()
Initializes query optimizer state.

There are 4 internals hash tables:

1. from type to declarations
2. from type to declarations for non-recursive queries
3. from type to name to declarations
4. from type to name to declarations for non-recursive queries

Almost every query includes declaration type information. Also very common query is to search some declaration(s) by name or full name. Those hash tables allows to search declaration very quick.

is_abstract
describes whether class abstract or not

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

mem_fun(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_function() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_funs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_functions() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_oper(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_operator() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_opers(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_operators() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

member_function(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member declaration, that is matched defined criteria

member_functions(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member function declarations, that are matched defined criteria

member_operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member operator declaration, that is matched defined criteria

member_operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member operator declarations, that are matched defined criteria

multiple_declarations_found_t

name
Declaration name @type: str

operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to operator declaration, that is matched defined criteria

operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of operator declarations, that are matched defined criteria

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

private_members
list of all private `members`

protected_members
list of all protected `members`

public_members
list of all public `members`

recursive_bases
list of all *base classes*

recursive_derived
list of all *derive classes*

remove_declaration(*decl*)
removes decl from members list

Parameters **decl** (`declaration_t`) – declaration to be removed

top_class
reference to a parent class, which contains this class and defined within a namespace

if this class is defined under a namespace, self will be returned

top_parent
Reference to top parent declaration.

@type: declaration_t

typedef(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to typedef declaration, that is matched defined criteria

typedefs(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of typedef declarations, that are matched defined criteria

use_demangled_as_name

variable(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to variable declaration, that is matched defined criteria

variables(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of variable declarations, that are matched defined criteria

class dependency_info_t(*decl*, *depend_on_it*, *access_type=None*, *hint=None*)
Bases: `object`

access_type

decl
Deprecated since 1.9.0. Will be removed in 2.0.0.

declaration

depend_on_it

find_out_depend_on_it_declarations()
If declaration depends on other declaration and not on some type this function will return reference to it. Otherwise None will be returned

hint
The declaration, that report dependency can put some additional inforamtion about dependency. It can be used later

static i_depend_on_them(*decl*)
Returns set of declarations. every item in the returned set, depends on a declaration from the input

static we_depend_on_them(*decls*)
Returns set of declarations. every item in the returned set, depends on a declaration from the input

get_partial_name(*name*)

class hierarchy_info_t(*related_class=None*, *access=None*, *is_virtual=False*)
Bases: `object`

describes class relationship

access

access_type
describes *hierarchy type*

declaration_path

declaration_path_hash

is_virtual
indicates whether the inheritance is virtual or not

related_class
reference to base or derived *class*

class impl_details
Bases: `object`

static dig_declarations(*depend_on_it*)

pygccxml.declarations.container_traits module

Define a few algorithms that deal with different properties of std containers.

all_container_traits = (<pygccxml.declarations.container_traits.container_traits_impl_t object>, <pygccxml.declaratio
tuple of all STD container traits classes

class container_traits_impl_t(*container_name*, *element_type_index*, *element_type_typedef*, *eraser*, *key_type_index=None*, *key_type_typedef=None*, *unordered_maps_and_sets=False*)
Bases: object

Implements the functionality needed for convenient work with STD container classes

Implemented functionality:

- find out whether a declaration is STD container or not
- find out container value(mapped) type

This class tries to be useful as much as possible. For example, for class declaration (and not definition) it parses the class name in order to extract the information.

class_declaration(*type_*)
Returns reference to the class declaration.

element_type(*type_*)
returns reference to the class valuemapped type declaration

get_container_or_none(*type_*)
Returns reference to the class declaration or None.

is_mapping(*type_*)

is_my_case(*type_*)
Checks, whether type is STD container or not.

is_sequence(*type_*)

key_type(*type_*)
returns reference to the class key type declaration

name()

remove_defaults(*type_or_string*)
Removes template defaults from a templated class instantiation.

For example:

```
std::vector< int, std::allocator< int > >
```

will become:

```
std::vector< int >
```

class defaults_eraser(*unordered_maps_and_sets*)
Bases: object

decorated_call_prefix(*cls_name*, *text*, *doit*)

decorated_call_suffix(*cls_name*, *text*, *doit*)

erase_allocator(*cls_name*, *default_allocator='std::allocator'*)

erase_call(*cls_name*)

erase_compare_allocator(*cls_name*, *default_compare='std::less'*, *default_allocator='std::allocator'*)

erase_container(*cls_name*, *default_container_name='std::deque'*)

erase_container_compare(*cls_name*, *default_container_name='std::vector'*, *default_compare='std::less'*)

erase_hash_allocator(*cls_name*)

erase_hashmap_compare_allocator(*cls_name*)

erase_map_compare_allocator(*cls_name*, *default_compare='std::less'*, *default_allocator='std::allocator'*)

erase_recursive(*cls_name*)

no_const(*cls_name*)

no_end_const(*cls_name*)

no_gnustd(*cls_name*)

no_std(*cls_name*)

no_stdext(*cls_name*)

normalize(*type_str*)

replace_basic_string(*cls_name*)

find_container_traits(*cls_or_string*)
Find the container traits type of a declaration.

Parameters **cls_or_string** (*str | declarations.declaration_t*) – a string

Returns a container traits

Return type *declarations.container_traits*

sequential_container_traits = [<pygccxml.declarations.container_traits.container_traits_impl_t object>, <pygccxml.d
list, that contains all STD container traits classes

pygccxml.declarations.cpptypes module

defines classes, that describe C++ types

FUNDAMENTAL_TYPES = {'short unsigned int': <pygccxml.declarations.cpptypes.short_unsigned_int_t object>, 'jdouble': <py
defines a mapping between fundamental type name and its synonym to the instance of class that describes the type

class array_t(*base*, *size*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents C++ array type

SIZE_UNKNOWN = -1

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

size
returns array size

class bool_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents bool type

CPPNAME = 'bool'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class calldef_type_t(*return_type=None*, *arguments_types=None*)
Bases: object

base class for all types that describes "callable" declaration

arguments_types
list of argument *types*

has_ellipsis

return_type
reference to *return type*

class char_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents char type

CPPNAME = 'char'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **complex_double_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents complex double type

CPPNAME = 'complex double'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **complex_float_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents complex float type

CPPNAME = 'complex float'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class complex_long_double_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents complex long double type

CPPNAME = 'complex long double'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class compound_t(*base*)
Bases: *pygccxml.declarations.cpptypes.type_t*

class that allows to represent compound types like *const int**

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **const_t**(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *whatever const* type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **declarated_t**(*declaration*)
Bases: *pygccxml.declarations.cpptypes.type_t*, *pygccxml.declarations.byte_info.byte_info*

class that binds between to hierarchies: *type_t* and `declaration_t`

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

declaration
reference to `declaration_t`

partial_decl_string

class double_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents double type

CPPNAME = 'double'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class dummy_type_t(*decl_string*)
Bases: *pygccxml.declarations.cpptypes.type_t*

provides *type_t* interface for a string, that defines C++ type.

This class could be very useful in the code generator.

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class elaborated_t(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *elaborated* type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class ellipsis_t
Bases: *pygccxml.declarations.cpptypes.type_t*

type, that represents "..." in function definition

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class float_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents float type

CPPNAME = 'float'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **free_function_type_t**(*return_type=None*, *arguments_types=None*)
Bases: *pygccxml.declarations.cpptypes.type_t*, *pygccxml.declarations.cpptypes.calldef_type_t*

describes free function type

NAME_TEMPLATE = '%(return_type)s (*)(%(arguments)s)'

TYPEDEF_NAME_TEMPLATE = '%(return_type)s (*%(typedef_name)s)(%(arguments)s)'

arguments_types
list of argument *types*

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

static **create_decl_string**(*return_type*, *arguments_types*, *with_defaults=True*)
Returns free function type

Parameters

- **return_type** (*type_t*) – function return type
- **arguments_types** – list of argument *type*

Return type *free_function_type_t*

create_typedef(*typedef_name*, *unused=None*, *with_defaults=True*)
returns string, that contains valid C++ code, that defines typedef to function type

Parameters **name** – the desired name of typedef

decl_string

has_ellipsis

partial_decl_string

return_type
reference to *return type*

class fundamental_t(*name*)
Bases: *pygccxml.declarations.cpptypes.type_t*

base class for all fundamental, build-in types

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class int128_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents __int128_t type

CPPNAME = '__int128_t'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class int_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents int type

CPPNAME = 'int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **java_fundamental_t**(*name*)
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

base class for all JNI defined fundamental types

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **jboolean_t**
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jboolean type

JNAME = 'jboolean'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **jbyte_t**
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jbyte type

JNAME = 'jbyte'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **jchar_t**
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jchar type

JNAME = 'jchar'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **jdouble_t**
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jdouble type

JNAME = 'jdouble'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **jfloat_t**
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jfloat type

JNAME = 'jfloat'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class jint_t
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jint type

JNAME = 'jint'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class jlong_t
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jlong type

JNAME = 'jlong'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class jshort_t
Bases: *pygccxml.declarations.cpptypes.java_fundamental_t*

represents jshort type

JNAME = 'jshort'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **long_double_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents long double type

CPPNAME = 'long double'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **long_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents long int type

CPPNAME = 'long int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **long_long_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents long long int type

CPPNAME = 'long long int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **long_long_unsigned_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents long long unsigned int type

CPPNAME = 'long long unsigned int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **long_unsigned_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents long unsigned int type

CPPNAME = 'long unsigned int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **member_function_type_t**(*class_inst=None*, *return_type=None*, *arguments_types=None*, *has_const=False*)
Bases: *pygccxml.declarations.cpptypes.type_t*, *pygccxml.declarations.cpptypes.calldef_type_t*

describes member function type

NAME_TEMPLATE = '%(return_type)s (%(class)s::*)(%(arguments)s)%(has_const)s'

TYPEDEF_NAME_TEMPLATE = '%(return_type)s (%(class)s::*%(typedef_name)s)(%(arguments)s) %(has_const)s'

arguments_types
list of argument *types*

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

class_inst
reference to parent `class`

clone()
returns new instance of the type

static create_decl_string(*return_type*, *class_decl_string*, *arguments_types*, *has_const*, *with_defaults=True*)

create_typedef(*typedef_name*, *class_alias=None*, *with_defaults=True*)
creates typedef to the function type

Parameters **typedef_name** – desired type name

Return type string

decl_string

has_const
describes, whether function has const modifier

has_ellipsis

partial_decl_string

return_type
reference to *return type*

class member_variable_type_t(*class_inst=None*, *variable_type=None*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

describes member variable type

NAME_TEMPLATE = '%(type)s (%(class)s::*)'

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

variable_type
describes member variable *type*

class pointer_t(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *whatever** type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **reference_t**(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *whatever&* type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **restrict_t**(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *restrict whatever* type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **short_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents short int type

CPPNAME = 'short int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **short_unsigned_int_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents short unsigned int type

CPPNAME = 'short unsigned int'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **signed_char_t**
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents signed char type

CPPNAME = 'signed char'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class **type_qualifiers_t**(*has_static=False*, *has_mutable=False*, *has_extern=False*)
Bases: object

contains additional information about type: mutable, static, extern

has_extern

has_mutable

has_static

class **type_t**
Bases: *pygccxml.declarations.byte_info.byte_info*

base class for all types

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class uint128_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents __uint128_t type

CPPNAME = '__uint128_t'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class unknown_t
Bases: *pygccxml.declarations.cpptypes.type_t*

type, that represents all C++ types, that could not be parsed by GCC-XML

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class unsigned_char_t

Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents unsigned char type

CPPNAME = 'unsigned char'

build_decl_string(*with_defaults=True*)

byte_align

Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size

Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()

returns new instance of the type

decl_string

partial_decl_string

class unsigned_int_t

Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents unsigned int type

CPPNAME = 'unsigned int'

build_decl_string(*with_defaults=True*)

byte_align

Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size

Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()

returns new instance of the type

decl_string

partial_decl_string

class void_t

Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents void type

CPPNAME = 'void'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class volatile_t(*base*)
Bases: *pygccxml.declarations.cpptypes.compound_t*

represents *volatile whatever* type

base
reference to internal/base class

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

class wchar_t
Bases: *pygccxml.declarations.cpptypes.fundamental_t*

represents wchar_t type

CPPNAME = 'wchar_t'

build_decl_string(*with_defaults=True*)

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

clone()
returns new instance of the type

decl_string

partial_decl_string

pygccxml.declarations.decl_factory module

defines default declarations factory class

class decl_factory_t
Bases: object

declarations factory class

create_casting_operator(**arguments*, ***keywords*)
creates instance of class that describes casting operator declaration

create_class(**arguments*, ***keywords*)
creates instance of class that describes class definition declaration

create_class_declaration(**arguments*, ***keywords*)
creates instance of class that describes class declaration

create_constructor(**arguments*, ***keywords*)
creates instance of class that describes constructor declaration

create_destructor(**arguments*, ***keywords*)
creates instance of class that describes destructor declaration

create_enumeration(**arguments*, ***keywords*)
creates instance of class that describes enumeration declaration

create_free_function(**arguments*, ***keywords*)
creates instance of class that describes free function declaration

create_free_operator(**arguments*, ***keywords*)
creates instance of class that describes free operator declaration

create_member_function(**arguments*, ***keywords*)
creates instance of class that describes member function declaration

create_member_operator(**arguments*, ***keywords*)
creates instance of class that describes member operator declaration

create_namespace(**arguments*, ***keywords*)
creates instance of class that describes namespace declaration

create_typedef(**arguments*, ***keywords*)
creates instance of class that describes typedef declaration

create_variable(**arguments*, ***keywords*)
creates instance of class that describes variable declaration

pygccxml.declarations.decl_printer module

defines class, *decl_printer_t* that prints declarations tree in a user friendly format

class decl_printer_t(*level=0*, *print_details=True*, *recursive=True*, *writer=<function _stdout_writer>*, *verbose=True*)
Bases: *pygccxml.declarations.decl_visitor.decl_visitor_t*

helper class for printing declarations tree

INDENT_SIZE = 4

JUSTIFY = 20

clone(*increment_level=True*)

instance

static is_builtin_decl(*decl*)

level

print_calldef_info(*decl=None*)

print_decl_header()

print_details

recursive

verbose

visit_casting_operator()

visit_class()

visit_class_declaration()

visit_constructor()

visit_destructor()

visit_enumeration()

visit_free_function()

visit_free_operator()

visit_member_function()

visit_member_operator()

visit_namespace()

visit_typedef()

visit_variable()

writer

dump_declarations(*declarations*, *file_path*)
Dump declarations tree rooted at each of the included nodes to the file

Parameters

- **declarations** – either a single :class:declaration_t object or a list of :class:declaration_t objects
- **file_path** – path to a file

print_declarations(*decls*, *detailed=True*, *recursive=True*, *writer=<function <lambda>>*, *verbose=True*)

print declarations tree rooted at each of the included nodes.

Parameters **decls** – either a single :class:declaration_t object or list of :class:declaration_t objects

pygccxml.declarations.decl_visitor module

defines declarations visitor class interface

class decl_visitor_t

Bases: `object`

declarations visitor interface

All functions within this class should be redefined in derived classes.

visit_casting_operator()

visit_class()

visit_class_declaration()

visit_constructor()

visit_destructor()

visit_enumeration()

visit_free_function()

visit_free_operator()

visit_member_function()

visit_member_operator()

visit_namespace()

visit_typedef()

visit_variable()

pygccxml.declarations.declaration module

Defines `pygccxml.declarations.declaration_t` class - all declarations base class.

class declaration_t(*name=''*, *location=None*, *is_artificial=False*, *mangled=None*, *demangled=None*, *attributes=None*)

Bases: `object`

Base class for all classes that represent a C++ declaration.

attributes

GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

get_mangled_name()

i_depend_on_them(*recursive=True*)
Return list of all types and declarations the declaration depends on

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

name
Declaration name @type: str

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

top_parent
Reference to top parent declaration.

@type: declaration_t

pygccxml.declarations.declaration_utils module

declaration_path(*decl*, *with_defaults=None*)
Returns a list of parent declarations names.

Parameters **decl** (declaration_t) – declaration for which declaration path should be calculated.

Returns

list of names, where first item is the top parent name and last item the inputted declaration name.

Return type list[(str | basestring)]

full_name(*decl*, *with_defaults=True*)
Returns declaration full qualified name.

If *decl* belongs to anonymous namespace or class, the function will return C++ illegal qualified name.

Parameters **decl** (declaration_t) – declaration for which the full qualified name should be calculated.

Returns full name of the declaration.

Return type list[(str | basestring)]

full_name_from_declaration_path(*dpath*)

get_named_parent(*decl*)
Returns a reference to a named parent declaration.

Parameters **decl** (declaration_t) – the child declaration

Returns the declaration or None if not found.

Return type *declaration_t*

partial_declaration_path(*decl*)
Returns a list of parent declarations names without template arguments that have default value.

Parameters **decl** (declaration_t) – declaration for which the partial declaration path should be calculated.

Returns

list of names, where first item is the top parent name and last item the inputted declaration name.

Return type list[(str | basestring)]

pygccxml.declarations.declarations_matchers module

class calldef_matcher_t(*name=None*, *return_type=None*, *arg_types=None*, *decl_type=None*, *header_dir=None*, *header_file=None*)
Bases: *pygccxml.declarations.declarations_matchers.declaration_matcher_t*

Instance of this class will match callable by the following criteria:

- *declaration_matcher_t* criteria
- return type. For example: int_t or 'int'
- argument types

check_name(*decl*)

decl_name_only

is_full_name()

name

class **declaration_matcher_t**(*name=None*, *decl_type=None*, *header_dir=None*, *header_file=None*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Instance of this class will match declarations by next criteria:

- declaration name, also could be fully qualified name Example: *wstring* or *::std::wstring*
- declaration type Example: `class_t`, `namespace_t`, `enumeration_t`
- location within file system (file or directory)

check_name(*decl*)

decl_name_only

is_full_name()

name

class **namespace_matcher_t**(*name=None*)
Bases: *pygccxml.declarations.declarations_matchers.declaration_matcher_t*

Instance of this class will match namespaces by name.

check_name(*decl*)

decl_name_only

is_full_name()

name

class **operator_matcher_t**(*name=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *decl_type=None*, *header_dir=None*, *header_file=None*)
Bases: *pygccxml.declarations.declarations_matchers.calldef_matcher_t*

Instance of this class will match operators by next criteria:

- *calldef_matcher_t* criteria
- operator symbol: =, !=, (), [] and etc

check_name(*decl*)

decl_name_only

is_full_name()

name

class **variable_matcher_t**(*name=None*, *decl_type=None*, *header_dir=None*, *header_file=None*)
Bases: *pygccxml.declarations.declarations_matchers.declaration_matcher_t*

Instance of this class will match variables by next criteria:

- *declaration_matcher_t* criteria
- variable type. Example: `int_t` or 'int'

check_name(*decl*)

decl_name_only

is_full_name()

name

pygccxml.declarations.elaborated_info module

class elaborated_info(*elaborated_type_specifier*)
Bases: object

This class stores the name of the elaborated type specifier.

elaborated_type_specifier
Elaborated specifier (can be – struct, union, class or enum).

Returns elaborated specifier

Return type str

pygccxml.declarations.enumeration module

defines class, that describes C++ *enum*

class enumeration_t(*name=''*, *values=None*)
Bases: *pygccxml.declarations.declaration.declaration_t*, *pygccxml.declarations.byte_info.byte_info*, *pygccxml.declarations.elaborated_info.elaborated_info*

describes C++ *enum*

append_value(*valuename*, *valuenum=None*)
Append another enumeration value to the *enum*.

The numeric value may be None in which case it is automatically determined by increasing the value of the last item.

When the 'values' attribute is accessed the resulting list will be in the same order as append_value() was called.

Parameters

- **valuename** (*str*) – The name of the value.
- **valuenum** (*int*) – The numeric value or None.

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

byte_align
Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size
Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

elaborated_type_specifier
Elaborated specifier (can be – struct, union, class or enum).

Returns elaborated specifier

Return type str

get_mangled_name()

get_name2value_dict()
returns a dictionary, that maps between *enum* name(key) and *enum* value(value)

has_value_name(*name*)
Check if this *enum* has a particular name among its values.

Parameters **name** (*str*) – Enumeration value name

Return type True if there is an enumeration value with the given name

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

name
Declaration name @type: str

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

top_parent
Reference to top parent declaration.

@type: declaration_t

values
A list of tuples (valname(str), valnum(int)) that contain the enumeration values. @type: list

pygccxml.declarations.free_calldef module

class free_calldef_t (**args*, ***keywords*)
Bases: *pygccxml.declarations.calldef.calldef_t*

base class for "callable" declarations that defined within C++ namespace

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string (*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()
This function should be overriden in the derived classes and return more-or-less successfull guess about calling convention

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

class free_function_t(**args*, ***keywords*)
Bases: *pygccxml.declarations.free_calldef.free_calldef_t*

describes free function declaration

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()
This function should be overriden in the derived classes and return more-or-less successfull guess about calling convention

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

top_parent
Reference to top parent declaration.

@type: declaration_t

class free_operator_t(**args*, ***keywords*)

Bases: *pygccxml.declarations.free_calldef.free_calldef_t*, *pygccxml.declarations.calldef_members.operator_t*

describes free operator declaration

OPERATOR_WORD_LEN = 8

argument_types
list of all argument types

arguments
The argument list. @type: list of `argument_t`

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calling_convention
function calling convention. See :class:CALLING_CONVENTION_TYPES class for possible values

class_types
list of class/class declaration types, extracted from the operator arguments

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

demangled_name
returns function demangled name. It can help you to deal with function template instantiations

does_throw
If False, than function does not throw any exception. In this case, function was declared with empty throw statement.

exceptions
The list of exceptions. @type: list of `declaration_t`

function_type()
returns function type. See `type_t` hierarchy

get_mangled_name()

guess_calling_convention()
This function should be overriden in the derived classes and return more-or-less successfull guess about calling convention

has_ellipsis

has_extern
Was this callable declared as "extern"? @type: bool

has_inline
Was this callable declared with "inline" specifier @type: bool

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

optional_args
list of all optional arguments, the arguments that have default value

overloads
A list of overloaded "callables" (i.e. other callables with the same name within the same scope.

@type: list of `calldef_t`

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

required_args
list of all required arguments

return_type
The type of the return value of the "callable" or None (constructors). @type: `type_t`

symbol
operator's symbol. For example – operator+, symbol is equal to '+'

top_parent
Reference to top parent declaration.

@type: declaration_t

pygccxml.declarations.function_traits module

defines few algorithms, that deals with different properties of functions

is_same_function(*f1*, *f2*)
returns true if f1 and f2 is same function

Use case: sometimes when user defines some virtual function in base class, it overrides it in a derived one. Sometimes we need to know whether two member functions is actually same function.

is_same_return_type(*f1*, *f2*)

pygccxml.declarations.has_operator_matcher module

has_public_binary_operator(*type_*, *operator_symbol*)
returns True, if *type_* has public binary operator, otherwise False

has_public_equal(*decl_type*)
returns True, if class has public operator==, otherwise False

has_public_less(*decl_type*)
returns True, if class has public operator<, otherwise False

pygccxml.declarations.location module

class location_t(*file_name=''*, *line=-1*)
Bases: `object`

Provides information about the location of the declaration within the source file.

as_tuple()
Return tuple(self.file_name, self.line)

file_name
Absolute source file name, type string.

line
Line number, type int.

pygccxml.declarations.matchers module

defines all "built-in" classes that implement declarations compare functionality according to some criteria

class access_type_matcher_t(*access_type*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Instance of this class will match declaration by its access type: public, private or protected. If declarations does not have access type, for example free function, then *False* will be returned.

class and_matcher_t(*matchers*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Combine several other matchers with "&" (and) operator.

For example: find all private functions with name XXX

```
matcher = access_type_matcher_t( 'private' ) &            calldef_matcher_t( name=
↪'XXX' )
```

class custom_matcher_t (*function*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Instance of this class will match declaration by user custom criteria.

class matcher_base_t
Bases: object

matcher_base_t class defines interface for classes that will implement compare functionality according to some criteria.

class not_matcher_t (*matcher*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

return the inverse result of a matcher

For example: find all public and protected declarations

```
matcher = ~access_type_matcher_t( 'private' )
```

class or_matcher_t (*matchers*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Combine several other matchers with "|" (or) operator.

For example: find all functions and variables with name 'XXX'

```
matcher = variable_matcher_t( name='XXX' ) |            calldef_matcher_t( name=
↪'XXX' )
```

class regex_matcher_t (*regex*, *function=None*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Instance of this class will match declaration using regular expression. User should supply a function that will extract from declaration desired information as string. Later, this matcher will match that string using user regular expression.

class virtuality_type_matcher_t (*virtuality_type*)
Bases: *pygccxml.declarations.matchers.matcher_base_t*

Instance of this class will match declaration by its virtual type: not virtual, virtual or pure virtual. If declarations does not have "virtual" property, for example free function, then *False* will be returned.

pygccxml.declarations.mdecl_wrapper module

defines class *mdecl_wrapper_t* that allows to work on set of declarations, as it was one declaration.

The *class* allows user to not write "for" loops within the code.

class call_redirector_t (*name*, *decls*)
Bases: object

Internal class used to call some function of objects

class mdecl_wrapper_t (*decls*)
Bases: object

multiple declarations class wrapper

The main purpose of this class is to allow an user to work on many declarations, as they were only one single declaration.

For example, instead of writing *for* loop like the following

```
for c in global_namespace.classes():
    c.attribute = "xxxx"
```

you can write:

```
global_namespace.classes().attribute = "xxxx"
```

The same functionality could be applied on "set" methods too.

to_list()

pygccxml.declarations.namespace module

Describe a C++ namespace declaration.

get_global_namespace(*decls*)
Get the global namespace (::) from a declaration tree.

Parameters **decls** (*list[declaration_t]*) – a list of declarations

Returns the global namespace_t object (::)

Return type *namespace_t*

class namespace_t(*name=''*, *declarations=None*)
Bases: *pygccxml.declarations.scopedef.scopedef_t*

Describes C++ namespace.

ALLOW_EMPTY_MDECL_WRAPPER = False

RECURSIVE_DEFAULT = True

adopt_declaration(*decl*)

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of algorithms_cache_t class.

calldef(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to "calldef" declaration, that is matched defined criteria

calldefs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of calldef_t declarations, that are matched defined criteria

casting_operator(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to casting operator declaration, that is matched defined criteria

casting_operators(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of casting operator declarations, that are matched defined criteria

class_(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to class declaration, that is matched defined criteria

classes(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of class declarations, that are matched defined criteria

clear_optimizer()
Cleans query optimizer state

constructor(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to constructor declaration, that is matched defined criteria

constructors(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of constructor declarations, that are matched defined criteria

create_decl_string(*with_defaults=True*)

decl(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to declaration, that is matched defined criteria

decl_string
Declaration full name.

declaration_not_found_t

declarations
List of children declarations.

Returns list[declaration_t]

decls(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of declarations, that are matched defined criteria

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

enum(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the enumeration() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

enumeration(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to enumeration declaration, that is matched defined criteria

enumerations(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of enumeration declarations, that are matched defined criteria

enums(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the enumerations() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

free_fun(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the free_function() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

free_function(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Returns reference to free function declaration that matches a defined criteria.

free_functions(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Returns a set of free function declarations that match a defined criteria.

free_funs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the free_functions() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

free_operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Returns reference to free operator declaration that matches a defined criteria.

free_operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Returns a set of free operator declarations that match a defined criteria.

get_mangled_name()

i_depend_on_them(*recursive=True*)

init_optimizer()
Initializes query optimizer state.

There are 4 internals hash tables:

1. from type to declarations
2. from type to declarations for non-recursive queries
3. from type to name to declarations
4. from type to name to declarations for non-recursive queries

Almost every query includes declaration type information. Also very common query is to search some declaration(s) by name or full name. Those hash tables allows to search declaration very quick.

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

mem_fun(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_function() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_funs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_functions() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_oper(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_operator() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_opers(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_operators() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

member_function(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member declaration, that is matched defined criteria

member_functions(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member function declarations, that are matched defined criteria

member_operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member operator declaration, that is matched defined criteria

member_operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member operator declarations, that are matched defined criteria

multiple_declarations_found_t

name
Declaration name @type: str

namespace(*name=None*, *function=None*, *recursive=None*)
Returns reference to namespace declaration that matches a defined criteria.

namespaces(*name=None*, *function=None*, *recursive=None*, *allow_empty=None*)
Returns a set of namespace declarations that match a defined criteria.

nss(*name=None*, *function=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the namespaces() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to operator declaration, that is matched defined criteria

operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of operator declarations, that are matched defined criteria

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

remove_declaration(*decl*)
Removes declaration from members list.

Parameters **decl** (declaration_t) – declaration to be removed

take_parenting(*inst*)
Takes parenting from inst and transfers it to self.

Parameters **inst** (namespace_t) – a namespace declaration

top_parent
Reference to top parent declaration.

@type: declaration_t

typedef(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to typedef declaration, that is matched defined criteria

typedefs(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of typedef declarations, that are matched defined criteria

variable(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to variable declaration, that is matched defined criteria

variables(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of variable declarations, that are matched defined criteria

pygccxml.declarations.pattern_parser module

Implementation details

class parser_t(*pattern_char_begin*, *pattern_char_end*, *pattern_char_separator*)
Bases: object

implementation details

NOT_FOUND = (-1, -1)
implementation details

args(*decl_string*)
Extracts a list of arguments from the provided declaration string.

Implementation detail. Example usages: Input: myClass<std::vector<int>, std::vector<double>> Output: [std::vector<int>, std::vector<double>]

Parameters **decl_string** (*str*) – the full declaration string

Returns list of arguments as strings

Return type list

find_args(*text*, *start=None*)
implementation details

has_pattern(*decl_string*)
Implementation detail

join(*name*, *args*, *arg_separator=None*)
implementation details

name(*decl_string*)
implementation details

normalize(*decl_string*, *arg_separator=None*)
implementation details

split(*decl_string*)
implementation details

split_recursive(*decl_string*)
implementation details

pygccxml.declarations.runtime_errors module

exception declaration_not_found_t(*decl_matcher*)
Bases: exceptions.RuntimeError

Exception raised when the declaration could not be found

args

message

exception multiple_declarations_found_t(*decl_matcher*)
Bases: exceptions.RuntimeError

Exception raised when more than one declaration was found

args

message

exception visit_function_has_not_been_found_t(*visitor*, *decl_inst*)
Bases: exceptions.RuntimeError

Exception that is raised, from apply_visitor(), when a visitor could not be applied.

args

message

pygccxml.declarations.scopedef module

Defines *scopedef_t* class

declaration_files(*decl_or_decls*)
Returns set of files

Every declaration is declared in some file. This function returns set, that contains all file names of declarations.

Parameters **decl_or_decls** (declaration_t or [declaration_t]) – reference to list of declaration's or single declaration

Return type set(declaration file names)

find_all_declarations(*declarations*, *decl_type=None*, *name=None*, *parent=None*, *recursive=True*, *fullname=None*)

Returns a list of all declarations that match criteria, defined by developer.

For more information about arguments see `match_declaration_t` class.

Return type [matched declarations]

find_declaration(*declarations*, *decl_type=None*, *name=None*, *parent=None*, *recursive=True*, *fullname=None*)

Returns single declaration that match criteria, defined by developer. If more the one declaration was found None will be returned.

For more information about arguments see `match_declaration_t` class.

Return type matched declaration `declaration_t` or None

find_first_declaration(*declarations*, *decl_type=None*, *name=None*, *parent=None*, *recursive=True*, *fullname=None*)

Returns first declaration that match criteria, defined by developer.

For more information about arguments see `match_declaration_t` class.

Return type matched declaration `declaration_t` or None

make_flatten(*decl_or_decls*)

Converts tree representation of declarations to flatten one.

Parameters **decl_or_decls** (`declaration_t` or [`declaration_t`]) – reference to list of declaration's or single declaration

Return type [all internal declarations]

class matcher

Bases: `object`

Class-namespace, contains implementation of a few "find" algorithms

static find(*decl_matcher*, *decls*, *recursive=True*)

Returns a list of declarations that match *decl_matcher* defined criteria or None

Parameters

- **decl_matcher** – Python callable object, that takes one argument - reference to a declaration
- **decls** – the search scope, :class:declaration_t object or :class:declaration_t objects list t
- **recursive** – boolean, if True, the method will run *decl_matcher* on the internal declarations too

static find_single(*decl_matcher*, *decls*, *recursive=True*)

Returns a reference to the declaration, that match *decl_matcher* defined criteria.

if a unique declaration could not be found the method will return None.

Parameters

- **decl_matcher** – Python callable object, that takes one argument - reference to a declaration
- **decls** – the search scope, :class:declaration_t object or :class:declaration_t objects list t
- **recursive** – boolean, if True, the method will run *decl_matcher* on the internal declarations too

static get_single(*decl_matcher*, *decls*, *recursive=True*)
Returns a reference to declaration, that match *decl_matcher* defined criteria.

If a unique declaration could not be found, an appropriate exception will be raised.

Parameters

- **decl_matcher** – Python callable object, that takes one argument - reference to a declaration
- **decls** – the search scope, :class:declaration_t object or :class:declaration_t objects list t
- **recursive** – boolean, if True, the method will run *decl_matcher* on the internal declarations too

class scopedef_t(*name=''*)
Bases: *pygccxml.declarations.declaration.declaration_t*

Base class for `namespace_t` and `class_t` classes.

This is the base class for all declaration classes that may have children nodes. The children can be accessed via the *scopedef_t.declarations* property.

Also this class provides "get/select/find" interface. Using this class you can get instance or instances of internal declaration(s).

You can find declaration(s) using next criteria:

1. *name* - declaration name, could be full qualified name
2. ***header_dir* - directory, to which belongs file, that the** declaration was declared in. *header_dir* should be absolute path.
3. *header_file* - file that the declaration was declared in.
4. ***function* - user (your) custom criteria. The interesting thing** is that this function will be joined with other arguments (criteria).
5. ***recursive* - the search declaration range, if True will be search** in internal declarations too.

Every ""query"" API, takes name or function as the first argument.

```
global_namespace.member_function("do_something)
```

the statement returns reference to member function named "do_something". If there the function doesn't exist or more than one function exists, an exception is raised.

If you want to query for many declarations, use other function(s):

```
do_something = global_namespace.member_functions("do_something")
```

the statement returns `mdecl_wrapper_t` instance. That object will save you writing *for* loops. For more information see `the class` documentation.

ALLOW_EMPTY_MDECL_WRAPPER = False

RECURSIVE_DEFAULT = True

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

calldef(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to "calldef" declaration, that is matched defined criteria

calldefs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of `calldef_t` declarations, that are matched defined criteria

casting_operator(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to casting operator declaration, that is matched defined criteria

casting_operators(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of casting operator declarations, that are matched defined criteria

class_(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to class declaration, that is matched defined criteria

classes(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of class declarations, that are matched defined criteria

clear_optimizer()
Cleans query optimizer state

constructor(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to constructor declaration, that is matched defined criteria

constructors(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of constructor declarations, that are matched defined criteria

create_decl_string(*with_defaults=True*)

decl(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to declaration, that is matched defined criteria

decl_string
Declaration full name.

declaration_not_found_t

declarations
List of children declarations.

Returns List[declarations.declaration_t]

decls(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of declarations, that are matched defined criteria

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

enum(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the enumeration() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

enumeration(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to enumeration declaration, that is matched defined criteria

enumerations(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of enumeration declarations, that are matched defined criteria

enums(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the enumerations() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

get_mangled_name()

i_depend_on_them(*recursive=True*)

init_optimizer()
Initializes query optimizer state.

There are 4 internals hash tables:

1. from type to declarations
2. from type to declarations for non-recursive queries
3. from type to name to declarations
4. from type to name to declarations for non-recursive queries

Almost every query includes declaration type information. Also very common query is to search some declaration(s) by name or full name. Those hash tables allows to search declaration very quick.

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

mem_fun(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_function() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_funs(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_functions() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_oper(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
Deprecated method. Use the member_operator() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

mem_opers(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
Deprecated method. Use the member_operators() method instead.

Deprecated since v1.9.0. Will be removed in v2.0.0

member_function(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member declaration, that is matched defined criteria

member_functions(*name=None*, *function=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member function declarations, that are matched defined criteria

member_operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to member operator declaration, that is matched defined criteria

member_operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of member operator declarations, that are matched defined criteria

multiple_declarations_found_t

name
Declaration name @type: str

operator(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to operator declaration, that is matched defined criteria

operators(*name=None*, *function=None*, *symbol=None*, *return_type=None*, *arg_types=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of operator declarations, that are matched defined criteria

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

remove_declaration(*decl*)

top_parent
Reference to top parent declaration.

@type: declaration_t

typedef(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to typedef declaration, that is matched defined criteria

typedefs(*name=None*, *function=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of typedef declarations, that are matched defined criteria

variable(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*)
returns reference to variable declaration, that is matched defined criteria

variables(*name=None*, *function=None*, *decl_type=None*, *header_dir=None*, *header_file=None*, *recursive=None*, *allow_empty=None*)
returns a set of variable declarations, that are matched defined criteria

pygccxml.declarations.smart_pointer_traits module

class auto_ptr_traits
Bases: object

implements functionality, needed for convenient work with *std::auto_ptr* pointers

static is_smart_pointer(*type_*)
returns True, if type represents instantiation of *boost::shared_ptr*, False otherwise

static value_type(*type_*)
returns reference to *boost::shared_ptr* value type

class internal_type_traits
Bases: object

small convenience class, which provides access to internal types

static get_by_name(*type_*, *name*)

class smart_pointer_traits
Bases: object

implements functionality, needed for convenient work with smart pointers

static is_smart_pointer(*type_*)
returns True, if type represents instantiation of *boost::shared_ptr* or *std::shared_ptr*, False otherwise

static value_type(*type_*)
returns reference to *boost::shared_ptr* or *std::shared_ptr* value type

pygccxml.declarations.templates module

template instantiation parser

This module provides functionality necessary to

- parse
- *split*
- *join*
- *normalize*

C++ template instantiations

args(*decl_string*)
returns list of template arguments

Return type [*str*]

is_instantiation(*decl_string*)
returns True if *decl_string* is template instantiation and False otherwise

Parameters **decl_string** (*str*) – string that should be checked for pattern presence

Return type bool

join(*name_*, *args_*)
returns name< argument_1, argument_2, ..., argument_n >

name(*decl_string*)
returns name of instantiated template

Return type str

normalize(*decl_string*)
returns *decl_string*, which contains "normalized" spaces

this functionality allows to implement comparison of 2 different string which are actually same: x::y< z > and x::y<z>

normalize_full_name_false(*decl*)
Cached variant of normalize

Parameters **decl** (declaration.declaration_t) – the declaration

Returns normalized name

Return type str

normalize_full_name_true(*decl*)
Cached variant of normalize

Parameters **decl** (declaration.declaration_t) – the declaration

Returns normalized name

Return type str

normalize_name(*decl*)
Cached variant of normalize

Parameters **decl** (declaration.declaration_t) – the declaration

Returns normalized name

Return type str

normalize_partial_name(*decl*)
Cached variant of normalize

Parameters **decl** (declaration.declaration_t) – the declaration

Returns normalized name

Return type str

split(*decl_string*)
returns (name, [arguments])

split_recursive(*decl_string*)
returns [(name, [arguments])]

pygccxml.declarations.traits_impl_details module

class impl_details
Bases: object

implementation details

static find_value_type(*global_ns*, *value_type_str*)
implementation details

static is_defined_in_xxx(*xxx*, *cls*)
Small helper method that checks whether the class *cls* is defined under *::xxx* namespace

pygccxml.declarations.type_traits module

defines few algorithms, that deals with different C++ type properties

Are you aware of boost::type_traits library? pygccxml implements the same functionality.

This module contains a set of very specific traits functionsclasses, each of which encapsulate a single trait from the C++ type system. For example: * is a type a pointer or a reference type ? * does a type have a trivial constructor ? * does a type have a const-qualifier ?

array_item_type(*type_*)
returns array item type

array_size(*type_*)
returns array size

base_type(*type_*)
returns base type.

For *const int* will return *int*

decompose_class(*type_*)
implementation details

decompose_type(*tp*)
Implementation detail

does_match_definition(*given*, *main*, *secondary*)
implementation details

is_arithmetic(*type_*)
returns True, if type represents C++ integral or floating point type, False otherwise

is_array(*type_*)
returns True, if type represents C++ array type, False otherwise

is_bool(*type_*)
Check if type is of boolean type.

Parameters **type** (type_t) – The type to be checked

Returns True if type is a boolean, False otherwise.

Return type bool

is_calldef_pointer(*type_*)
returns True, if type represents pointer to free/member function, False otherwise

is_const(*type_*)
returns True, if type represents C++ const type, False otherwise

is_elaborated(*type_*)
returns True, if type represents C++ elaborated type, False otherwise

is_floating_point(*type_*)
returns True, if type represents C++ floating point type, False otherwise

is_fundamental(*type_*)
returns True, if type represents C++ fundamental type

is_integral(*type_*)
Check if type is a C++ integral type

Parameters **type** (type_t) – The type to be checked

Returns True if type is a C++ integral type, False otherwise.

Return type bool

is_pointer(*type_*)
returns True, if type represents C++ pointer type, False otherwise

is_reference(*type_*)
returns True, if type represents C++ reference type, False otherwise

is_same(*type1*, *type2*)
returns True, if type1 and type2 are same types

is_std_ostream(*type_*)
Returns True, if type represents C++ std::ostream, False otherwise.

is_std_string(*type_*)
Returns True, if type represents C++ *std::string*, False otherwise.

is_std_wostream(*type_*)
Returns True, if type represents C++ std::wostream, False otherwise.

is_std_wstring(*type_*)
Returns True, if type represents C++ *std::wstring*, False otherwise.

is_void(*type_*)
Check if type is of void type.

Parameters **type** (type_t) – The type to be checked

Returns True if type is void, False otherwise.

Return type bool

is_void_pointer(*type_*)
returns True, if type represents *void**, False otherwise

is_volatile(*type_*)
returns True, if type represents C++ volatile type, False otherwise

remove_alias(*type_*)
Returns *type_t* without typedef

Parameters **type** (*type_t | declaration_t*) – type or declaration

Returns the type associated to the inputted declaration

Return type *type_t*

remove_const(*type_*)
removes const from the type definition

If type is not const type, it will be returned as is

remove_cv(*type_*)
removes const and volatile from the type definition

remove_declarated(*type_*)
removes type-declaration class-binder `declarated_t` from the *type_*

If *type_* is not `declarated_t`, it will be returned as is

remove_elaborated(*type_*)
removes type-declaration class-binder `elaborated_t` from the *type_*

If *type_* is not `elaborated_t`, it will be returned as is

remove_pointer(*type_*)
removes pointer from the type definition

If type is not pointer type, it will be returned as is.

remove_reference(*type_*)
removes reference from the type definition

If type is not reference type, it will be returned as is.

remove_volatile(*type_*)
removes volatile from the type definition

If type is not volatile type, it will be returned as is

pygccxml.declarations.type_traits_classes module

class_declaration_traits = <pygccxml.declarations.type_traits_classes.declaration_xxx_traits object>
implements functionality, needed for convenient work with C++ class declarations

class_traits = <pygccxml.declarations.type_traits_classes.declaration_xxx_traits object>
implements functionality, needed for convenient work with C++ classes

class **declaration_xxx_traits**(*declaration_class*)
Bases: `object`

this class implements the functionality needed for convenient work with declaration classes

Implemented functionality:

- find out whether a declaration is a desired one
- get reference to the declaration

get_declaration(*type_*)
returns reference to the declaration

Precondition: self.is_my_case(type) == True

is_my_case(*type_*)
returns True, if type represents the desired declaration, False otherwise

enum_declaration = <bound method declaration_xxx_traits.get_declaration of <pygccxml.declarations.type_traits_classes.
returns reference to enum declaration

enum_traits = <pygccxml.declarations.type_traits_classes.declaration_xxx_traits object>
implements functionality, needed for convenient work with C++ enums

find_copy_constructor(*type_*)
Returns reference to copy constructor.

Parameters **type** (*declarations.class_t*) – the class to be searched.

Returns the copy constructor

Return type declarations.constructor_t

find_noncopyable_vars(*class_type*, *already_visited_cls_vars=None*)
Returns list of all *noncopyable* variables.

If an already_visited_cls_vars list is provided as argument, the returned list will not contain these variables. This list will be extended with whatever variables pointing to classes have been found.

Parameters

- **class_type** (*declarations.class_t*) – the class to be searched.
- **already_visited_cls_vars** (*list*) – optional list of vars that should not be checked a second time, to prevent infinite recursions.

Returns list of all *noncopyable* variables.

Return type list

find_trivial_constructor(*type_*)
Returns reference to trivial constructor.

Parameters **type** (*declarations.class_t*) – the class to be searched.

Returns the trivial constructor

Return type declarations.constructor_t

has_any_non_copyconstructor(*decl_type*)
if class has any public constructor, which is not copy constructor, this function will return list of them, otherwise None

has_copy_constructor(*class_*)
if class has public copy constructor, this function will return reference to it, None otherwise

has_destructor(*class_*)
if class has destructor, this function will return reference to it, None otherwise

has_public_assign(*class_*)
returns True, if class has public assign operator, False otherwise

has_public_constructor(*class_*)
if class has any public constructor, this function will return list of them, otherwise None

has_public_destructor(*decl_type*)
returns True, if class has public destructor, False otherwise

has_trivial_constructor(*class_*)
if class has public trivial constructor, this function will return reference to it, None otherwise

has_vtable(*decl_type*)
True, if class has virtual table, False otherwise

is_base_and_derived(*based*, *derived*)
returns True, if there is "base and derived" relationship between classes, False otherwise

is_binary_operator(*oper*)
returns True, if operator is binary operator, otherwise False

is_class = <bound method declaration_xxx_traits.is_my_case of <pygccxml.declarations.type_traits_classes.declaration_xxx
returns True, if type represents C++ class definition, False otherwise

is_class_declaration = <bound method declaration_xxx_traits.is_my_case of <pygccxml.declarations.type_traits_classes
returns True, if type represents C++ class declaration, False otherwise

is_convertible(*source*, *target*)
returns True, if source could be converted to target, otherwise False

is_copy_constructor(*constructor*)
Check if the declaration is a copy constructor,

Parameters **constructor** (*declarations.constructor_t*) – the constructor to be checked.

Returns True if this is a copy constructor, False instead.

Return type bool

is_enum = <bound method declaration_xxx_traits.is_my_case of <pygccxml.declarations.type_traits_classes.declaration_xxx_
returns True, if type represents C++ enumeration declaration, False otherwise

is_noncopyable(*class_*, *already_visited_cls_vars=None*)
Checks if class is non copyable

Parameters

- **class** (*declarations.class_t*) – the class to be checked
- **already_visited_cls_vars** (*list*) – optional list of vars that should not be checked a second time, to prevent infinite recursions. In general you can ignore this argument, it is mainly used during recursive calls of is_noncopyable() done by pygccxml.

Returns if the class is non copyable

Return type bool

is_struct(*declaration*)
Returns True if declaration represents a C++ struct

Parameters **declaration** (declaration_t) – the declaration to be checked.

Returns True if declaration represents a C++ struct

Return type bool

is_trivial_constructor(*constructor*)
Check if the declaration is a trivial constructor.

Parameters **constructor** (*declarations.constructor_t*) – the constructor to be checked.

Returns True if this is a trivial constructor, False instead.

Return type bool

is_unary_operator(*oper*)
returns True, if operator is unary operator, otherwise False

is_union(*declaration*)
Returns True if declaration represents a C++ union

Parameters **declaration** (declaration_t) – the declaration to be checked.

Returns True if declaration represents a C++ union

Return type bool

pygccxml.declarations.type_visitor module

defines types visitor class interface

class type_visitor_t

Bases: object

types visitor interface

All functions within this class should be redefined in derived classes.

visit_array()

visit_bool()

visit_char()

visit_complex_double()

visit_complex_float()

visit_complex_long_double()

visit_const()

visit_declarated()

visit_double()

visit_elaborated()

visit_ellipsis()

visit_float()

visit_free_function_type()

visit_int()

visit_int128()

visit_jboolean()

visit_jbyte()

visit_jchar()

visit_jdouble()

visit_jfloat()

visit_jint()

visit_jlong()

visit_jshort()

visit_long_double()

visit_long_int()

visit_long_long_int()

visit_long_long_unsigned_int()

visit_long_unsigned_int()

visit_member_function_type()

visit_member_variable_type()

visit_pointer()

visit_reference()

visit_restrict()

visit_short_int()

visit_short_unsigned_int()

visit_signed_char()

visit_uint128()

visit_unsigned_char()

visit_unsigned_int()

visit_void()

visit_volatile()

visit_wchar()

pygccxml.declarations.typedef module

defines class that describes C++ typedef declaration

class typedef_t(*name=''*, *decl_type=None*)

Bases: *pygccxml.declarations.declaration.declaration_t*, *pygccxml.declarations.byte_info.byte_info*

describes C++ typedef declaration

attributes

GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

byte_align

Alignment of this declaration/type in bytes

Returns Alignment of this declaration/type in bytes

Return type int

byte_size

Size of this declaration/type in bytes

Returns Size of this declaration/type in bytes

Return type int

cache

Implementation detail.

Reference to instance of `algorithms_cache_t` class.

create_decl_string(*with_defaults=True*)

decl_string

Declaration full name.

decl_type

reference to the original `decl_type`

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

get_mangled_name()

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

For GCCXML, you can get the mangled name for all the declarations. When using CastXML, calling mangled is only allowed on functions and variables. For other declarations it will raise an exception.

Returns the mangled name

Return type str

name
Declaration name @type: str

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

top_parent
Reference to top parent declaration.

@type: declaration_t

pygccxml.declarations.variable module

defines class that describes C++ global and member variable declaration

class variable_t(*name=''*, *decl_type=None*, *type_qualifiers=None*, *value=None*, *bits=None*, *mangled=None*)
Bases: *pygccxml.declarations.declaration.declaration_t*

describes C++ global and member variable declaration

access_type

attributes
GCCXML attributes, set using __attribute__((gccxml("...")))

@type: str

bits
integer, that contains information about how many bit takes bit field

byte_offset
integer, offset of the field from the beginning of class.

cache
Implementation detail.

Reference to instance of `algorithms_cache_t` class.

create_decl_string(*with_defaults=True*)

decl_string
Declaration full name.

decl_type
reference to the variable `decl_type`

decorated_name
Unique declaration name extracted from a binary file (.map, .dll, .so, etc).

@type: str

demangled
Declaration name, reconstructed from GCCXML generated unique name.

@type: str

get_mangled_name()

i_depend_on_them(*recursive=True*)

is_artificial
Describes whether declaration is compiler generated or not

@type: bool

location
Location of the declaration within source file

@type: `location_t`

mangled
Unique declaration name generated by the compiler.

Returns the mangled name

Return type str

name
Declaration name @type: str

parent
Reference to parent declaration.

@type: declaration_t

partial_decl_string
Declaration full name.

partial_name
Declaration name, without template default arguments.

Right now std containers is the only classes that support this functionality.

top_parent
Reference to top parent declaration.

@type: declaration_t

type_qualifiers
reference to the `type_qualifiers_t` instance

value
string, that contains the variable value

pygccxml.declarations.xml_generators module

Contains enumeration of all xml_generators supported by the project.

on_missing_functionality(*xml_generator*, *functionality*)

pygccxml.parser package

Parser sub-package.

parse(*files*, *config=None*, *compilation_mode='file by file'*, *cache=None*)
Parse header files.

Parameters

- **files** (*list of str*) – The header files that should be parsed
- **config** (`parser.xml_generator_configuration_t`) – Configuration object or None
- **compilation_mode** (`parser.COMPILATION_MODE`) – Determines whether the files are parsed individually or as one single chunk
- **cache** (`parser.cache_base_t` or str) – Declaration cache (None=no cache)

Return type list of `declarations.declaration_t`

parse_string(*content*, *config=None*)

parse_xml_file(*content*, *config=None*)

Submodules

pygccxml.parser.config module

Defines C++ parser configuration classes.

create_compiler_path(*xml_generator*, *compiler_path*)
Try to guess a path for the compiler.

If you want ot use a specific compiler, please provide the compiler path manually, as the guess may not be what you are expecting. Providing the path can be done by passing it as an argument (compiler_path) to the xml_generator_configuration_t() or by defining it in your pygccxml configuration file.

load_xml_generator_configuration(*configuration*, ***defaults*)
Loads CastXML or GCC-XML configuration.

Parameters

- **configuration** (*string|configparser.ConfigParser*) – can be a string (file path to a configuration file) or instance of `configparser.ConfigParser`.
- **defaults** – can be used to override single configuration values.

Returns a configuration object

Return type *xml_generator_configuration_t*

The file passed needs to be in a format that can be parsed by `configparser.ConfigParser`.

An example configuration file skeleton can be found here.

class **parser_configuration_t**(*working_directory='.'*, *include_paths=None*, *define_symbols=None*, *undefine_symbols=None*, *cflags=''*, *compiler=None*, *xml_generator=None*, *keep_xml=False*, *compiler_path=None*, *flags=None*, *castxml_epic_version=None*)
Bases: `object`

C++ parser configuration holder

This class serves as a base class for the parameters that can be used to customize the call to a C++ parser.

This class also allows users to work with relative files paths. In this case files are searched in the following order:

1.current directory

2.working directory

3.additional include paths specified by the user

append_cflags(*val*)

castxml_epic_version
File format version used by castxml.

cflags
additional flags to pass to compiler

clone()

compiler
get compiler name to simulate

compiler_path
Get the path for the compiler.

define_symbols
list of "define" directives

flags
Optional flags for pygccxml.

include_paths
list of include paths to look for header files

keep_xml
Are xml files kept after errors.

raise_on_wrong_settings()
Validates the configuration settings and raises RuntimeError on error

undefine_symbols
list of "undefine" directives

working_directory

xml_generator
get xml_generator (gccxml or castxml)

class **xml_generator_configuration_t**(*gccxml_path=''*, *xml_generator_path=''*, *working_directory='.'*, *include_paths=None*, *define_symbols=None*, *undefine_symbols=None*, *start_with_declarations=None*, *ignore_gccxml_output=False*, *cflags=''*, *compiler=None*, *xml_generator=None*, *keep_xml=False*, *compiler_path=None*, *flags=None*, *castxml_epic_version=None*)
Bases: *pygccxml.parser.config.parser_configuration_t*

Configuration object to collect parameters for invoking gccxml or castxml.

This class serves as a container for the parameters that can be used to customize the call to gccxml or castxml.

append_cflags(*val*)

castxml_epic_version
File format version used by castxml.

cflags
additional flags to pass to compiler

clone()

compiler
get compiler name to simulate

compiler_path
Get the path for the compiler.

define_symbols
list of "define" directives

flags
Optional flags for pygccxml.

ignore_gccxml_output
set this property to True, if you want pygccxml to ignore any error warning that comes from gccxml

include_paths
list of include paths to look for header files

keep_xml
Are xml files kept after errors.

raise_on_wrong_settings()

start_with_declarations
list of declarations gccxml should start with, when it dumps declaration tree

undefine_symbols
list of "undefine" directives

working_directory

xml_generator
get xml_generator (gccxml or castxml)

xml_generator_from_xml_file
Configuration object containing information about the xml generator read from the xml file.

Returns configuration object

Return type *utils.xml_generators*

xml_generator_path
XML generator binary location

pygccxml.parser.declarations_cache module

class cache_base_t
Bases: `object`

cached_value(*source_file*, *configuration*)
Return declarations, we have cached, for the source_file and the given configuration.

Parameters

- **source_file** – path to the C++ source file being parsed.
- **configuration** – configuration that was used for parsing.

flush()
Flush (write out) the cache to disk if needed.

logger = <logging.Logger object>

update(*source_file*, *configuration*, *declarations*, *included_files*)
update cache entry

Parameters

- **source_file** – path to the C++ source file being parsed
- **configuration** – configuration used in parsing `xml_generator_configuration_t`
- **declarations** – declaration tree found when parsing
- **included_files** – files included by parsing.

configuration_signature(*config*)
Return a signature for a configuration (xml_generator_configuration_t) object.

This can then be used as a key in the cache. This method must take into account anything about a configuration that could cause the declarations generated to be different between runs.

class dummy_cache_t
Bases: *pygccxml.parser.declarations_cache.cache_base_t*

This is an empty cache object.

By default no caching is enabled in pygccxml.

cached_value(*source_file*, *configuration*)

flush()

logger = <logging.Logger object>

update(*source_file*, *configuration*, *declarations*, *included_files*)

class file_cache_t (*name*)
Bases: *pygccxml.parser.declarations_cache.cache_base_t*

Cache implementation to store data in a pickled form in a file. This class contains some cache logic that keeps track of which entries have been 'hit' in the cache and if an entry has not been hit then it is deleted at the time of the flush(). This keeps the cache from growing larger when files change and are not used again.

cached_value (*source_file*, *configuration*)
Attempt to lookup the cached declarations for the given file and configuration.

Returns None if declaration not found or signature check fails.

flush ()

logger = <logging.Logger object>

update (*source_file*, *configuration*, *declarations*, *included_files*)
Update a cached record with the current key and value contents.

file_signature (*filename*)
Return a signature for a file.

class record_t (*source_signature*, *config_signature*, *included_files*, *included_files_signature*, *declarations*)
Bases: object

config_signature

static create_key (*source_file*, *configuration*)

declarations

included_files

included_files_signature

key ()

source_signature

was_hit

pygccxml.parser.declarations_joiner module

bind_aliases (*decls*)
This function binds between class and it's typedefs.

Parameters **decls** – list of all declarations

Return type None

join_declarations (*namespace*)

pygccxml.parser.directory_cache module

directory cache implementation.

This module contains the implementation of a cache that uses individual files, stored in a dedicated cache directory, to store the cached contents.

The parser.directory_cache_t class instance could be passed as the *cache* argument of the parser.parse() function.

class directory_cache_t (*dir='cache'*, *directory='cache'*, *compression=False*, *sha1_sigs=True*)
Bases: *pygccxml.parser.declarations_cache.cache_base_t*

cache class that stores its data as multiple files inside a directory.

The cache stores one index file called *index.dat* which is always read by the cache when the cache object is created. Each header file will have its corresponding .cache file that stores the declarations found in the header file. The index file is used to determine whether a .cache file is still valid or not (by checking if one of the dependent files (i.e. the header file itself and all included files) have been modified since the last run).

cached_value (*source_file*, *configuration*)
Return the cached declarations or None.

Parameters

- **source_file** (*str*) – Header file name
- **configuration** (parser.xml_generator_configuration_t) – Configuration object

Return type Cached declarations or None

flush ()
Save the index table to disk.

logger = <logging.Logger object>

update (*source_file*, *configuration*, *declarations*, *included_files*)
Replace a cache entry by a new value.

Parameters

- **source_file** (*str*) – a C++ source file name.
- **configuration** (xml_generator_configuration_t) – configuration object.
- **declarations** (*pickable object*) – declarations contained in the *source_file*
- **included_files** (*list of str*) – included files

class filename_entry_t (*filename*)
Bases: object

This is a record stored in the filename_repository_t class.

The class is an internal class used in the implementation of the filename_repository_t class and it just serves as a container for the file name and the reference count.

dec_ref_count ()
Decrease the reference count by 1 and return the new count.

inc_ref_count ()
Increase the reference count by 1.

class filename_repository_t (*sha1_sigs*)
Bases: object

File name repository.

This class stores file names and can check whether a file has been modified or not since a previous call. A file name is stored by calling acquire_filename() which returns an ID and a signature of the file. The signature can later be used to check if the file was modified by calling is_file_modified(). If the file name is no longer required release_filename() should be called so that the entry can be removed from the repository.

acquire_filename (*name*)
Acquire a file name and return its id and its signature.

is_file_modified(*id_*, *signature*)
Check if the file referred to by *id_* has been modified.

release_filename(*id_*)
Release a file name.

update_id_counter()
Update the *id_* counter so that it doesn't grow forever.

class **index_entry_t**(*filesigs*, *configsig*)
Bases: `object`

Entry of the index table in the directory cache index.

Each cached header file (i.e. each .cache file) has a corresponding index_entry_t object. This object is used to determine whether the cache file with the declarations is still valid or not.

This class is a helper class for the directory_cache_t class.

pygccxml.parser.etree_scanner module

class **ietree_scanner_t**(*xml_file*, *decl_factory*, **args*)
Bases: *pygccxml.parser.scanner.scanner_t*

access()

calldefs()

characters(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

declarations()

endDocument()

endElement(*name*)

endElementNS(*name*, *qname*)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS event.

endPrefixMapping(*prefix*)
End the scope of a prefix-URI mapping.

See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed.

enums()

files()

ignorableWhitespace(*whitespace*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

members()

processingInstruction(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

read()

setDocumentLocator(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

skippedEntity(*name*)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the http://xml.org/sax/features/external-general-entities and the http://xml.org/sax/features/external-parameter-entities properties.

startDocument()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).

startElement(*name*, *attrs*)

startElementNS(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

The uri part of the name tuple is None for elements which have no namespace.

startPrefixMapping(*prefix*, *uri*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the http://xml.org/sax/features/namespaces feature is true (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.

Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not guaranteed.

types()

xml_generator_from_xml_file
Configuration object containing information about the xml generator read from the xml file.

Returns configuration object

Return type *utils.xml_generators*

pygccxml.parser.linker module

class linker_t(*decls*, *types*, *access*, *membership*, *files*, *xml_generator_from_xml_file=None*)
Bases: *pygccxml.declarations.decl_visitor.decl_visitor_t*, *pygccxml.declarations.type_visitor.type_visitor_t*, object

instance

visit_array()

visit_bool()

visit_casting_operator()

visit_char()

visit_class()

visit_class_declaration()

visit_complex_double()

visit_complex_float()

visit_complex_long_double()

visit_const()

visit_constructor()

visit_declarated()

visit_destructor()

visit_double()

visit_elaborated()

visit_ellipsis()

visit_enumeration()

visit_float()

visit_free_function()

visit_free_function_type()

visit_free_operator()

visit_int()

visit_int128()

visit_jboolean()

visit_jbyte()

visit_jchar()

visit_jdouble()

visit_jfloat()

visit_jint()

visit_jlong()

visit_jshort()

visit_long_double()

visit_long_int()

visit_long_long_int()

visit_long_long_unsigned_int()

visit_long_unsigned_int()

visit_member_function()

visit_member_function_type()

visit_member_operator()

visit_member_variable_type()

visit_namespace()

visit_pointer()

visit_reference()

visit_restrict()

visit_short_int()

visit_short_unsigned_int()

visit_signed_char()

visit_typedef()

visit_uint128()

visit_unsigned_char()

visit_unsigned_int()

visit_variable()

visit_void()

visit_volatile()

visit_wchar()

pygccxml.parser.patcher module

class casting_operator_patcher_t
Bases: object

class default_argument_patcher_t (*enums*, *cxx_std*)
Bases: object

fix_calldef_decls (*decls*, *enums*, *cxx_std*)
some times gccxml report typedefs defined in no namespace it happens for example in next situation template< typename X> void ddd(){ typedef typename X::Y YY;} if I will fail on this bug next time, the right way to fix it may be different

update_unnamed_class (*decls*)
Adds name to class_t declarations.

If CastXML is being used, the type definitions with an unnamed class/struct are split across two nodes in the XML tree. For example,

typedef struct {} cls;

produces

<Struct id="_7" name="" context="_1" .../> <Typedef id="_8" name="cls" type="_7" context="_1" .../>

For each typedef, we look at which class it refers to, and update the name accordingly. This helps the matcher classes finding these declarations. This was the behaviour with gccxml too, so this is important for backward compatibility.

If the castxml epic version 1 is used, there is even an elaborated type declaration between the typedef and the struct/class, that also needs to be taken care of.

Parameters **decls** (*list[declaration_t]*) – a list of declarations to be patched.

Returns None

pygccxml.parser.project_reader module

class COMPILATION_MODE
Bases: object

ALL_AT_ONCE = 'all at once'

FILE_BY_FILE = 'file by file'

create_cached_source_fc (*header*, *cached_source_file*)
Creates parser.file_configuration_t instance, configured to contain path to GCC-XML generated XML file and C++ source file. If XML file does not exists, it will be created and used for parsing. If XML file exists, it will be used for parsing.

Parameters

- **header** (*str*) – path to C++ source file
- **cached_source_file** (*str*) – path to GCC-XML generated XML file

Return type parser.file_configuration_t

create_gccxml_fc (*xml_file*)
Creates parser.file_configuration_t instance, configured to contain path to GCC-XML generated XML file.

Parameters **xml_file** (*str*) – path to GCC-XML generated XML file

Return type parser.file_configuration_t

create_source_fc(*header*)
Creates parser.file_configuration_t instance, configured to contain path to C++ source file

Parameters **header** (*str*) – path to C++ source file

Return type parser.file_configuration_t

create_text_fc(*text*)
Creates parser.file_configuration_t instance, configured to contain Python string, that contains valid C++ code

Parameters **text** (*str*) – C++ code

Return type parser.file_configuration_t

class file_configuration_t(*data*, *start_with_declarations=None*, *content_type='standard source file'*, *cached_source_file=None*)
Bases: object

source code location configuration.

The class instance uses "variant" interface to represent the following data:

1. path to a C++ source file
2. path to GCC-XML generated XML file
3. path to a C++ source file and path to GCC-XML generated file

   In this case, if XML file does not exists, it will be created. Next time you will ask to parse the source file, the XML file will be used instead.

   Small tip: you can setup your makefile to delete XML files every time, the relevant source file was changed.
4. Python string, that contains valid C++ code

There are few functions, that will help you to construct *file_configuration_t* object:

- *create_source_fc()*
- *create_gccxml_fc()*
- *create_cached_source_fc()*
- *create_text_fc()*

class CONTENT_TYPE
Bases: object

CACHED_SOURCE_FILE = 'cached source file'

GCCXML_GENERATED_FILE = 'gccxml generated file'

STANDARD_SOURCE_FILE = 'standard source file'

TEXT = 'text'

file_configuration_t.**cached_source_file**

file_configuration_t.**content_type**

file_configuration_t.**data**

file_configuration_t.**start_with_declarations**

class **project_reader_t**(*config*, *cache=None*, *decl_factory=None*)
Bases: object

parses header files and returns the contained declarations

static **get_os_file_names**(*files*)
returns file names

Parameters **files** (*list*) – list of strings andor *file_configuration_t* instances.

read_files(*files*, *compilation_mode='file by file'*)
parses a set of files

Parameters

- **files** (*list*) – list of strings andor *file_configuration_t* instances.
- **compilation_mode** (*COMPILATION_MODE*) – determines whether the files are parsed individually or as one single chunk

Return type [declaration_t]

read_string(*content*)
Parse a string containing C/C++ source code.

Parameters **content** (*str*) – C/C++ source code.

Return type Declarations

read_xml(*file_configuration*)
parses C++ code, defined on the file_configurations and returns GCCXML generated file content

xml_generator_from_xml_file
Configuration object containing information about the xml generator read from the xml file.

Returns configuration object

Return type *utils.xml_generators*

pygccxml.parser.scanner module

class **scanner_t**(*xml_file*, *decl_factory*, *config*, **args*)
Bases: xml.sax.handler.ContentHandler

access()

calldefs()

characters(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

declarations()

endDocument()

endElement(*name*)

endElementNS(*name*, *qname*)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS event.

endPrefixMapping(*prefix*)
End the scope of a prefix-URI mapping.

See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed.

enums()

files()

ignorableWhitespace(*whitespace*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

members()

processingInstruction(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

read()

setDocumentLocator(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

skippedEntity(*name*)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the http://xml.org/sax/features/external-general-entities and the http://xml.org/sax/features/external-parameter-entities properties.

startDocument()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).

startElement(*name*, *attrs*)

startElementNS(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

The uri part of the name tuple is None for elements which have no namespace.

startPrefixMapping(*prefix*, *uri*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the http://xml.org/sax/features/namespaces feature is true (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.

Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not guaranteed.

types()

xml_generator_from_xml_file
Configuration object containing information about the xml generator read from the xml file.

Returns configuration object

Return type *utils.xml_generators*

pygccxml.parser.source_reader module

bind_aliases(*decls*)
This function binds between class and it's typedefs.

Deprecated since 1.9.0, will be removed in 2.0.0

Parameters **decls** – list of all declarations

Return type None

class source_reader_t(*configuration*, *cache=None*, *decl_factory=None*)
Bases: `object`

This class reads C++ source code and returns the declarations tree.

This class is the only class that works directly with GCC-XML or CastXML.

It has only one responsibility: it calls GCC-XML with a source file specified by the user and creates declarations tree. The implementation of this class is split to two classes:

1. ***scanner_t* - this class scans the "XML" file, generated by GCC-XML** or CastXML and creates *pygccxml* declarations and types classes. After the XML file has been processed declarations and type class instances keeps references to each other using GCC-XML or CastXML generated id's.
2. ***linker_t* - this class contains logic for replacing GCC-XML or CastXML** generated ids with references to declarations or type class instances.

create_xml_file(*source_file*, *destination=None*)
This method will generate a xml file using an external tool.

The external tool can be either gccxml or castxml. The method will return the file path of the generated xml file.

Parameters

- **source_file** (*str*) – path to the source file that should be parsed.
- **destination** (*str*) – if given, will be used as target file path for GCC-XML or CastXML.

Return type path to xml file.

create_xml_file_from_string(*content*, *destination=None*)
Creates XML file from text.

Parameters

- **content** (*str*) – C++ source code
- **destination** (*str*) – file name for GCC-XML generated file

Return type returns file name of GCC-XML generated file

join_declarations(*namespace*)

read_cpp_source_file(*source_file*)
Reads C++ source file and returns declarations tree

Parameters **source_file** (*str*) – path to C++ source file

read_file(*source_file*)

read_string(*content*)
Reads a Python string that contains C++ code, and return the declarations tree.

read_xml_file(*xml_file*)
Read generated XML file.

Parameters **xml_file** (*str*) – path to xml file

Return type declarations tree

xml_generator_from_xml_file
Configuration object containing information about the xml generator read from the xml file.

Returns configuration object

Return type *utils.xml_generators*

pygccxml.utils package

The utils package contains tools used internally by pygccxml.

Submodules

pygccxml.utils.utils module

Logger classes and a few convenience methods.

class **DeprecationWrapper**(*new_target*, *old_name*, *new_name*, *version*)
Bases: `object`

A small wrapper class useful when deprecation classes.

This class is not part of the public API.

class **cached**(*method*)
Bases: `property`

Convert a method into a cached attribute.

deleter()
Descriptor to change the deleter on a property.

fdel

fget

fset

getter()
Descriptor to change the getter on a property.

reset()

setter()
Descriptor to change the setter on a property.

contains_parent_dir(*fpath*, *dirs*)
Returns true if paths in dirs start with fpath.

Precondition: dirs and fpath should be normalized before calling this function.

create_temp_file_name(*suffix*, *prefix=None*, *dir=None*, *directory=None*)
Small convenience function that creates temporary files.

This function is a wrapper around the Python built-in function tempfile.mkstemp.

class **cxx_standard**(*cflags*)
Bases: `object`

Helper class for parsing the C++ standard version.

This class holds the C++ standard version the XML generator has been configured with, and provides helpers functions for querying C++ standard version related information.

is_cxx03
Returns true if -std=c++03 is being used

is_cxx11
Returns true if -std=c++11 is being used

is_cxx11_or_greater
Returns true if -std=c++11 or a newer standard is being used

is_cxx14
Returns true if -std=c++14 is being used

is_cxx14_or_greater
Returns true if -std=c++14 or a newer standard is being used

is_cxx1z
Returns true if -std=c++1z is being used

is_implicit
Indicates whether a -std=c++xx was specified

stdcxx
Returns the -std=c++xx option passed to the constructor

find_xml_generator(*name=None*)
Try to find a c++ parser. Returns path and name.

Parameters **name** (*str*) – name of the c++ parser: castxml or gccxml

If no name is given the function first looks for castxml, then for gccxml. If no c++ parser is found the function raises an exception.

get_architecture()
Returns computer architecture: 32 or 64.

The guess is based on maxint.

get_tr1(*name*)
In libstd++ the tr1 namespace needs special care.

Return either an empty string or tr1::, useful for appending to search patterns.

Parameters **name** (*str*) – the name of the declaration

Returns an empty string or "tr1::"

Return type str

is_str(*string*)
Python 2 and 3 compatible string checker.

Parameters **string** (*str | basestring*) – the string to check

Returns True or False

Return type bool

class loggers
Bases: object

Class-namespace, defines a few loggers classes, used in the project.

all_loggers = [<logging.Logger object>, <logging.Logger object>, <logging.Logger object>, <logging.Logger object>, <
Contains all logger classes, defined by the class.

cxx_parser = <logging.Logger object>
Logger for C++ parser functionality

If you set this logger level to DEBUG, you will be able to see the exact command line, used to invoke GCC-XML and errors that occures during XML parsing

declarations_cache = <logging.Logger object>
Logger for declarations tree cache functionality

If you set this logger level to DEBUG, you will be able to see what is exactly happens, when you read the declarations from cache file. You will be able to decide, whether it worse for you to use this or that cache strategy.

pdb_reader = <logging.Logger object>
Logger for MS .pdb file reader functionality

queries_engine = <logging.Logger object>
Logger for query engine functionality.

If you set this logger level to DEBUG, you will be able to see what queries you do against declarations tree, measure performance and may be even to improve it. Query engine reports queries and whether they are optimized or not.

root = <logging.Logger object>
Root logger exists for your convenience only.

static **set_level**(*level*)
Set the same logging level for all the loggers at once.

normalize_path(*some_path*)
Return os.path.normpath(os.path.normcase(some_path)).

remove_file_no_raise(*file_name*, *config*)
Removes file from disk if exception is raised.

pygccxml.utils.xml_generators module

class **xml_generators**(*logger*, *gccxml_cvs_revision=None*, *castxml_format=None*)
Bases: `object`

Helper class for the xml generator versions

get_string_repr()
Get a string identifier for the current type of xml generator

Returns identifier

Return type str

is_castxml
Is the current xml generator castxml?

Returns is castxml being used?

Return type bool

is_castxml1
Is the current xml generator castxml (with output format version 1)?

Returns is castxml (with output format version 1) being used?

Return type bool

is_gccxml
Is the current xml generator gccxml?

Returns is gccxml being used?

Return type bool

is_gccxml_06
Is the current xml generator gccxml (version 0.6)?

Returns is gccxml 0.6 being used?

Return type bool

is_gccxml_07
Is the current xml generator gccxml (version 0.7)?

Returns is gccxml 0.7 being used?

Return type bool

is_gccxml_09
Is the current xml generator gccxml (version 0.9)?

Returns is gccxml 0.9 being used?

Return type bool

is_gccxml_09_buggy
Is the current xml generator gccxml (version 0.9 - buggy)?

Returns is gccxml 0.9 (buggy) being used?

Return type bool

xml_output_version
The current xml output version for the parsed file.

Returns the xml output version

Return type str

Building the documentation

Building the documentation locally

You can build the documentation yourself. In order for this to work you need sphinx doc (http://sphinx-doc.org) and the readthedocs theme:

pip install sphinx

pip install sphinx_rtd_theme

Then just run the following command in the root folder:

make html

This will build the documentation locally in the *docs/_build/html* folder.

For each commit on the master and develop branches, the documentation is automatically built and can be found here: https://readthedocs.org/projects/pygccxml/

Declarations query API

Introduction

You parsed the source files. Now you have to do some real work with the extracted information, right? pygccxml provides very powerful and simple interface to query about extracted declarations.

Just an example. I want to select all member functions, which have 2 arguments. I don't care about first argument type, but I do want second argument type to be a reference to an integer. More over, I want those functions names to end with "impl" string and they should be protected or private.

```
#global_ns is the reference to an instance of namespace_t object, that
#represents global namespace
query = declarations.custom_matcher_t( lambda mem_fun: mem_fun.name.endswith( 'impl' )
query = query & ~declarations.access_type_matcher_t( 'public' )
global_ns.member_functions( function=query, arg_types=[None, 'int &'] )
```

The example is complex, but still readable. In many cases you will find yourself, looking for one or many declarations, using one or two declaration properties. For example:

```
global_ns.namespaces( 'details' )
```

This call will return all namespaces with name 'details'.

User interface

As you already know, `pygccxml.declarations` package defines the following classes:

- *`scopedef_t`* - base class for all classes, that can contain other declarations
- *`namespace_t`* - derives from *`scopedef_t`* class, represents C++ namespace
- *`class_t`* - derives from *`scopedef_t`* class, represents C++ class/struct/union.

So, the query methods defined on `scopedef_t` class could be used on instances of `class_t` and `namespace_t` classes. I am sure you knew that.

Usage examples

I will explain the usage of `member_function` and `member_functions` methods. The usage of other methods is very similar to them. Here is definition of those methods:

```
def member_function(  self,
                      name=None,
                      function=None,
                      return_type=None,
                      arg_types=None,
                      header_dir=None,
                      header_file=None,
                      recursive=None )

mem_fun = member_function #just an alias

def member_functions( self,
                      name=None,
                      function=None,
                      return_type=None,
                      arg_types=None,
                      header_dir=None,
                      header_file=None,
                      recursive=None,
                      allow_empty=None )
mem_funs = member_functions
```

As you can see, from the method arguments you can search for member function by:

- `name`

Python string, that contains member function name or full name.

```
do_smth = my_class.member_function( 'do_smth' )
do_smth = my_class.member_function( 'my_namespace::my_class::do_smth' )
```

- `function`

  Python callable object. You would use this functionality, if you need to build custom query. This object will be called with one argument - declaration, and it should return `True` or `False`.

  ```
  impls = my_class.member_functions( lambda decl: decl.name.endswith( 'impl' ) )
  ```

  `impls` will contain all member functions, that their name ends with "impl".

- `return_type`

  the function return type. This argument can be string or an object that describes C++ type.

  ```
  mem_funcs = my_class.member_functions( return_type='int' )

  i = declarations.int_t()
  ref_i = declarations.reference_t( i )
  const_ref_i = declarations.const_t( ref_i )
  mem_funcs = my_class.member_functions( return_type=const_ref_int )
  ```

- `arg_types`

  Python list that contains description of member function argument types. This list could be a mix of Python strings and objects that describes C++ type. Size of list says how many arguments function should have. If you want to skip some argument type from within comparison, you put `None`, into relevant position within the list.

  ```
  mem_funcs = my_class.member_functions( arg_types=[ None, 'int'] )
  ```

  `mem_funcs` will contain all member functions, which have two arguments and type of second argument is `int`.

- `header_dir`

  Python string, that contains full path to directory, which contains file, which contains the function declaration

  ```
  mem_funcs = my_namespace.member_functions( header_dir='/home/roman/xyz' )
  ```

- `header_file`

  Python string, that contains full path to file, which contains the function declaration.

  ```
  mem_funcs = my_namespace.member_functions( header_dir='/home/roman/xyz/
  xyz.hpp' )
  ```

- `recursive`

  Python boolean object.

  If `recursive` is `True`, then member function will be also searched within internal declarations.

  If `recursive` is `False`, then member function will be searched only within current scope.

  What happen if `recursive` is `None`? Well. `scopedef_t` class defines `RECURSIVE_DEFAULT` variable. Its initial value is `True`. So, if you don't pass `recursive` argument, the value of `RECURSIVE_DEFAULT` variable will be used. This "yet another level of indirection" allows you to configure pygccxml "select" functions in one place for all project.

- `allow_empty`

  Python boolean object, it says pygccxml what to do if query returns empty.

  If `allow_empty` is `False`, then exception `RuntimeError( "Multi declaration query returned 0 declarations." )` will be raised

  `allow_empty` uses same technique as `recursive`, to allow you to customize the behavior project-wise. The relevant class variable name is `ALLOW_EMPTY_MDECL_WRAPPER`. Its initial value is `False`.

Now, when you understand, how to call those functions, I will explain what they return.

`member_function` will always return reference to desired declaration. If declaration could not be found or there are more then one declaration that match query `RuntimeError` exception will be raised.

Return value of `member_functions` is not Python list or set, but instance of `mdecl_wrapper_t` class. This class allows you to work on all selected objects at once. I will give an example from another project - https://pypi.python.org/pypi/pyplusplus/. In order to help Boost.Python to manage objects life time, all functions should have call policies. For example:

```
struct A{
    A* clone() const { return new A(); }
    ...
};
```

```
struct B{
    B* clone() const { return new B(); }
    ...
};
```

`clone` member function call policies is `return_value_policy<manage_new_object>()`. The following code applies the call policies on all `clone` member functions within the project:

```
#global_ns - instance of namespace_t class, that contains reference to global␣
↪namespace
clone = global_ns.member_functions( 'clone' )
clone.call_policies = return_value_policy( manage_new_object )
```

Another example, from https://pypi.python.org/pypi/pyplusplus/ project. Sometimes it is desirable to exclude declaration, from being exported to Python. The following code will exclude `clone` member function from being exported:

```
global_ns.member_functions( 'clone' ).exclude()
```

As you can see this class allows you to write less code. Basically using this class you don't have to write loops. If will do it for you. Also if you insist to write loops, `mdecl_wrapper_t` class implements `__len__`, `__getitem__` and `__iter__` methods. So you can write the following code:

```
for clone in global_ns.member_functions( 'clone' ):
    print clone.parent.name
```

Implementation details

Performance

For big projects, performance is critical. When you finished to build/change declarations tree, then you can call `scopedef_t.init_optimizer` method. This method will initialize few data structures, that will help to minimize the number of compared declarations. The price you are going to pay is memory usage.

Data structures

Here is a short explanation of what data structures is initialized.

- `scopedef_t._type2decls`, `scopedef_t._type2decls_nr`

  Python dictionary, that contains mapping between declaration type and declarations in the current scope.

  `scopedef_t.type2decls_nr` contains only declaration from the current scope. `scopedef_t.type2decls` contains declarations from the current scope and its children

- `scopedef_t._type2name2decls`, `scopedef_t._type2name2decls_nr`

  Python dictionary, that contains mapping between declaration type and another dictionary. This second dictionary contains mapping between a declaration name and declaration.

  `scopedef_t.type2name2decls_nr` contains only declaration from the current scope. `scopedef_t.type2name2decls` contains declarations from the current scope and its children

- `scopedef_t._all_decls`

  A flat list of all declarations, including declarations from the children scopes.

Except `scopedef_t.decl` and `scopedef_t.decls` methods, all other queries have information about declaration type.

If you include `name` into your query, you will get the best performance.

More information

I think, I gave you the important information. If you need definition of some query method, you can take a look on API documentation or into source code.

Design overview

pygccxml has 4 packages:

- *`declarations`*

  This package defines classes that describe C++ declarations and types.

- *`parser`*

  This package defines classes that parse GCC-XML or CastXML generated files. It also defines a few classes that will help you unnecessary parsing of C++ source files.

- *`utils`*

  This package defines a few functions useful for the whole project, but which are mainly used internally by pygccxml.

`declarations` package

Please take a look on the UML diagram. This UML diagram describes almost all classes defined in the package and their relationship. `declarations` package defines two hierarchies of class:

1. types hierarchy - used to represent a C++ type
2. declarations hierarchy - used to represent a C++ declaration

Types hierarchy

Types hierarchy is used to represent an arbitrary type in C++. class `type_t` is the base class.

type_traits

Are you aware of boost::type_traits library? The boost::type_traits library contains a set of very specific traits classes, each of which encapsulate a single trait from the C++ type system; for example, is a type a pointer or a reference? Or does a type have a trivial constructor, or a const-qualifier?

pygccxml implements a lot of functionality from the library:

- a lot of algorithms were implemented
  - `is_same`
  - `is_enum`
  - `is_void`
  - `is_const`
  - `is_array`
  - `is_pointer`
  - `is_volatile`
  - `is_integral`
  - `is_reference`
  - `is_arithmetic`
  - `is_convertible`
  - `is_fundamental`
  - `is_floating_point`
  - `is_base_and_derived`
  - `is_unary_operator`
  - `is_binary_operator`
  - `remove_cv`
  - `remove_const`
  - `remove_alias`
  - `remove_pointer`
  - `remove_volatile`
  - `remove_reference`
  - `has_trivial_copy`
  - `has_trivial_constructor`
  - `has_any_non_copyconstructor`

  For a full list of implemented algorithms, please consult API documentation.
- a lot of unit tests has been written base on unit tests from the boost::type_traits library.

If you are going to build code generator, you will find `type_traits` very handy.

Declarations hierarchy

A declaration hierarchy is used to represent an arbitrary C++ declaration. Basically, most of the classes defined in this package are just "set of properties".

`declaration_t` is the base class of the declaration hierarchy. Every declaration has `parent` property. This property keeps a reference to the scope declaration instance, in which this declaration is defined.

The `scopedef_t` class derives from `declaration_t`. This class is used to say - "I may have other declarations inside". The "composite" design pattern is used here. `class_t` and `namespace_t` declaration classes derive from the `scopedef_t` class.

`parser` package

Please take a look on parser package UML diagram . Classes defined in this package, implement parsing and linking functionality. There are few kind of classes defined by the package:

- classes, that implements parsing algorithms of GCC-XML generated XML file
- parser configuration classes
- cache - classes, those one will help you to eliminate unnecessary parsing
- patchers - classes, which fix GCC-XML generated declarations. (Yes, sometimes GCC-XML generates wrong description of C++ declaration.)

Parser classes

`source_reader_t` - the only class that have a detailed knowledge about GCC-XML. It has only one responsibility: it calls GCC-XML with a source file specified by user and creates declarations tree. The implementation of this class is split to 2 classes:

1. `scanner_t` - this class scans the "XML" file, generated by GCC-XML and creates pygccxml declarations and types classes. After the xml file has been processed declarations and type class instances keeps references to each other using GCC-XML generated ids.
2. `linker_t` - this class contains logic for replacing GCC-XML generated ids with references to declarations or type class instances.

Both those classes are implementation details and should not be used by user. Performance note: `scanner_t` class uses Python `xml.sax` package in order to parse XML. As a result, `scanner_t` class is able to parse even big XML files pretty quick.

`project_reader_t` - think about this class as a linker. In most cases you work with few source files. GCC-XML does not supports this mode of work. So, pygccxml implements all functionality needed to parse few source files at once. `project_reader_t` implements 2 different algorithms, that solves the problem:

1. `project_reader_t` creates temporal source file, which includes all the source files.
2. `project_reader_t` parse separately every source file, using `source_reader_t` class and then joins the resulting declarations tree into single declarations tree.

Both approaches have different trades-off. The first approach does not allow you to reuse information from already parsed source files. While the second one allows you to setup cache.

Parser configuration classes

`gccxml_configuration_t` - a class, that accumulates all the settings needed to invoke GCC-XML:

`file_configuration_t` - a class, that contains some data and description how to treat the data. `file_configuration_t` can contain reference to the the following types of data:

1. path to C++ source file
2. path to GCC-XML generated XML file
3. path to C++ source file and path to GCC-XML generated XML file

   In this case, if XML file does not exists, it will be created. Next time you will ask to parse the source file, the XML file will be used instead.

   Small tip: you can setup your makefile to delete XML files every time, the relevant source file has changed.
4. Python string, that contains valid C++ code

There are few functions that will help you to construct `file_configuration_t` object:

- `def create_source_fc( header )`

  `header` contains path to C++ source file
- `def create_gccxml_fc( xml_file )`

  `xml_file` contains path to GCC-XML generated XML file
- `def create_cached_source_fc( header, cached_source_file )`
  - `header` contains path to C++ source file
  - `xml_file` contains path to GCC-XML generated XML file
- `def create_text_fc( text )`

  `text` - Python string, that contains valid C++ code

Cache classes

There are few cache classes, which implements different cache strategies.

1. `file_configuration_t` class, that keeps path to C++ source file and path to GCC-XML generated XML file.
2. `file_cache_t` class, will save all declarations from all files within single binary file.
3. `directory_cache_t` class will store one index file called "index.dat" which is always read by the cache when the cache object is created. Each header file will have its corresponding *.cache file that stores the declarations found in the header file. The index file is used to determine whether a *.cache file is still valid or not (by checking if one of the dependent files (i.e. the header file itself and all included files) have been modified since the last run).

In some cases, `directory_cache_t` class gives much better performance, than `file_cache_t`. Many thanks to Matthias Baas for its implementation.

Warning: when pygccxml writes information to files, using cache classes, it does not write any version information. It means, that when you upgrade pygccxml you have to delete all your cache files. Otherwise you will get very strange errors. For example: missing attribute.

Patchers

Well, GCC-XML has few bugs, which could not be fixed from it. For example

```
namespace ns1{ namespace ns2{
    enum fruit{ apple, orange };
} }
```

```
void fix_enum( ns1::ns2::fruit arg=ns1::ns2::apple );
```

GCC-XML will report the default value of `arg` as `apple`. Obviously this in an error. pygccxml knows how to fix this bug.

This is not the only bug, which could be fixed, there are few of them. pygccxml introduces few classes, which knows how to deal with specific bug. More over, those bugs are fixed, only if I am 101% sure, that this is the right thing to do.

`utils` package

Use internally by pygccxml. Some methods/classes may be still usefull: loggers, find_xml_generator

Summary

That's all. I hope I was clear, at least I tried. Any way, pygccxml is an open source project. You always can take a look on the source code. If you need more information please read API documentation.

Who is using pygccxml?

Users

- The Insight Toolkit is using pygccxml (http://www.itk.org).
- PyBindGen - is a Python module that is geared to generating C/C++ code that binds a C/C++ library for Python.
- your project name ... :-)

pygccxml in blogs

- http://blog.susheelspace.com/?p=88

  " ... I have used pygccxml for parsing c++ code, it was a lot of fun to use "
- http://cysquatch.net/blog/2007/09/01/c-code-metrics-with-pygccxml

  pygccxml is used to calculate the Weighted Methods per Class (WMC) metric.
- http://www.garagegames.com/blogs/4280/13907

  pygccxml is used to generate input files for SIP code generator.
- http://blogs.sun.com/thorsten/entry/more_on_source_code_grokking

  Short listing of C++ parsers and their description.

C++ Reflection

Links

- CppReflect - extracts reflection information from executables, which were build with debug information. It works with executables that has COFF or ELF format.
- XTI An Extended Type Information Library - Bjarne Stroustrup talk about adding reflection information to C++ program.

Releasing

To build a new release, modify the version number in:

`pygccxml/__init__.py`

This version number will then automatically be used to build the documentation and by the setup.py script when building the wheels.

Do not forget to document the changes in the `CHANGELOG.md` file.

Uploading to pypi

The documentation for the building and uploading can be found here: pypi

The wheels are built with:

```
python setup.py bdist_wheel --universal
```

They are uploaded with:

```
twine upload dist/*
```

History and Credits

History

The original author and maintainer for pygccxml was Roman Yakovenko (2004-2011).

Holger Frydrych forked the project to work on python 3 support. Finally, Mark Moll forked the project a second time to carry on the work of porting the code to python 3 (keeping it compatible with python 2).

In Mai 2014, Michka Popoff and the Insight Software Consortium revived pygccxml by setting up a git repositery on GitHub, hosted along with gccxml.

The full changelog can be found in the CHANGELOG.md file.

Contributors

Thanks to all the people that have contributed patches, bug reports and suggestions, or supported this project.

- Roman Yakovenko (original author)
- Michka Popoff (actual maintainer)

A special thanks to the Insight Software Consortium for their help and collaboration, especially:

- Brad King
- Matt Mccormick

Contributors can be found on github's contributors page: https://github.com/gccxml/pygccxml/graphs/contributors

Here is the original list of contributors from before the GitHub migration.

- Roman Yakovenko's wife - Yulia
- Mark Moll
- Holger Frydrych
- John Pallister
- Matthias Baas
- Allen Bierbaum
- Georgiy Dernovoy
- Darren Garnier
- Gottfried Ganssauge
- Gaetan Lehmann
- Martin Preisler
- Miguel Lobo
- Jeremy Sanders
- Ben Schleimer
- Gustavo Carneiro
- Christopher Bruns
- Alejandro Dubrovsky
- Aron Xu

GCC-XML 0.7 → 0.9 upgrade issues (Legacy)

Introduction

This page is kept here for historical reasons. CastXML is the recommended xml generator and GCC-XML is being phased out. This page will be removed in version 2.0.0 of pygcxml.

Recently, GCC-XML internal parser was updated to GCC 4.2. The internal representation of source code, provided by GCC's parser, has changed a lot and few backward compatibility issues were introduced. In this document, I will try to cover all problems you may encounter.

Default constructor

GCC-XML 0.9 doesn't report compiler generated default and copy constructors as an implicit ones.

If you rely heavily on their existence in the generated XML, I suggest you to switch to `has_trivial_constructor` and to `has_trivial_copy` functions.

Pointer to member variable

Previous version of GCC-XML reported pointer to member variable as a "PointerType" with reference to "OffsetType". The new version removes "PointerType" from this sequence.

C++ code:

```
struct xyz_t{
  int do_smth( double );
  int m_some_member;
};

typedef int (xyz_t::*mfun_ptr_t)( double );

typedef int (xyz_t::*mvar_ptr_t);
```

GCC-XML 0.7:

```
<Typedef id="_6" name="mfun_ptr_t" type="_5" />
<PointerType id="_5" type="_128" size="32" align="32"/>
<MethodType id="_128" basetype="_7" returns="_136">
  <Argument type="_140"/>
</MethodType>

<Typedef id="_4" name="mvar_ptr_t" type="_3" />
<PointerType id="_3" type="_127" size="32" align="32"/>
<OffsetType id="_127" basetype="_7" type="_136" size="32" align="32"/>
```

GCC-XML 0.9:

```
<Typedef id="_97" name="mfun_ptr_t" type="_96" />
<PointerType id="_96" type="_147" size="32" align="32"/>
<MethodType id="_147" basetype="_92" returns="_131">
  <Argument type="_127"/>
</MethodType>

<Typedef id="_52" name="mvar_ptr_t" type="_139" />
<OffsetType id="_139" basetype="_92" type="_131" size="32" align="32"/>
```

pygccxml handles this issue automatically, you don't have to change your code.

Constant variable value

GCC-XML 0.9 uses suffix to report the constant variable value

For example:

```
const long unsigned int initialized = 10122004;
```

GCC-XML 0.9 will report the `initialized` value as `10122004ul`, while GCC-XML 0.7 as `10122004`.

pygccxml handles this problem automatically, you don't have to change your code.

Free and member function default arguments

Both versions of GCC-XML have a few issues, related to default arguments. GCC-XML 0.9 fixes some issues, but introduces another ones. Take a look on the following examples:

- Example 1

```
void fix_numeric( ull arg=(ull)-1 );
```

GCC-XML 0.7

```
<Argument name="arg" type="_7" default="0xffffffffffffffff"/>
```

GCC-XML 0.9

```
<Argument name="arg" type="_103" default="0xffffffffffffffffu"/>
```

- Example 2

```
void fix_function_call( int i=calc( 1,2,3) );
```

GCC-XML 0.7

```
<Argument name="i" type="_9" default="function_call::calc(int, int, int)(1, 2, 3)
↪"/>
```

GCC-XML 0.9

```
<Argument name="i" type="_34" default="function_call::calc(1, 2, 3)"/>
```

- Example 3

```
void typedef__func( const typedef_::alias& position = typedef_::alias() );
```

GCC-XML 0.7

```
<Argument name="position" type="_1458" default="alias()"/>
```

GCC-XML 0.9

```
<Argument name="position" type="_1703" default="typedef_::original_name()"/>
```

- Example 4

```
void typedef__func2( const typedef_::alias& position = alias() );
```

GCC-XML 0.7

```
<Argument name="position" type="_1458" default="alias()"/>
```

GCC-XML 0.9

```
<Argument name="position" type="_1703" default="typedef_::original_name()"/>
```

- Example 5

```
node* clone_tree( const std::vector<std::string> &types=std::vector<std::string>
↪() );
```

GCC-XML 0.7

```
<Argument name="types" type="_3336" default="vector&lt;std::basic_string&lt;char,␣
↪std::char_traits&lt;char&gt;, std::allocator&lt;char&gt; &gt;,std::allocator&lt;
↪std::basic_string&lt;char, std::char_traits&lt;char&gt;, std::allocator&lt;char&
↪gt; &gt; &gt; &gt;((&amp;allocator&lt;std::basic_string&lt;char, std::char_
↪traits&lt;char&gt;, std::allocator&lt;char&gt; &gt; &gt;()))"/>
```

GCC-XML 0.9

```
<Argument name="types" type="_3096" default="std::vector&lt;std::basic_string&lt;
↪char, std::char_traits&lt;char&gt;, std::allocator&lt;char&gt; &gt;,␣
↪std::allocator&lt;std::basic_string&lt;char, std::char_traits&lt;char&gt;,␣
↪std::allocator&lt;char&gt; &gt; &gt; &gt;(((const std::allocator&lt;std::basic_
↪string&lt;char, std::char_traits&lt;char&gt;, std::allocator&lt;char&gt; &gt; &
↪gt;&amp;)((const std::allocator&lt;std::basic_string&lt;char, std::char_traits&
↪lt;char&gt;, std::allocator&lt;char&gt; &gt; &gt;*)(&amp; std::allocator&lt;
↪std::basic_string&lt;char, std::char_traits&lt;char&gt;, std::allocator&lt;char&
↪gt; &gt; &gt;()))))"/>
```

Basically pygccxml can't help you here. The good news is that you always can change the default value expression from the script:

```
#f is "calldef_t" instance
for arg in f.arguments:
    arg.default_value = <<<new default value or None>>>
```

Name mangling

GCC-XML 0.9 mangles names different than the previous one. This change is the most dramatic one, because it may require from you to change the code.

Consider the following C++ code:

```
template< unsigned long i1>
struct item_t{
  static const unsigned long v1 = i1;
};

struct buggy{
  typedef unsigned long ulong;
  typedef item_t< ulong( 0xDEECE66DUL ) | (ulong(0x5) << 32) > my_item_t;
  my_item_t my_item_var;
};
```

generated data	GCC-XML 0.7	GCC-XML 0.9
class name	item_t<0x0deece66d>	item_t<-554899859ul>
class mangled name	6item_tILm3740067437EE	6item_tILm3740067437EE
class demangled name	item_t<3740067437l>	item_t<3740067437ul>

pygccxml uses class demangled name as a "name" of the class. This was done to overcome few bugs GCC-XML has, when it works on libraries with extreme usage of templates.

As you can see the name of the class is different. pygccxml is unable to help you in such situations. I suggest you to use query API strict mode. This is the default one. If the class/declaration with the given name could not be found, it will raise an error with clear description of the problem.

You can also to print the declarations tree to `stdout` and find out the name of the class/declaration from it.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X