Parsing C++ with GCC Plugins

Boris Kolpackov

Code Synthesis

v1.1, May 2011

GNU Compiler Collection

Who has never heard of GCC?

Object-Relational Mapping for C++

Who has never heard of ODB?

GCC Plugin Architecture

- Dynamic loading
- Hook into compilation pipeline *anywhere*
- ...starting from compiler startup
- ...ending with assembler output

```
g++ -fplugin=name -fplugin-arg-name-key[=value]
```

GCC with Plugin Support

- Available since GCC 4.5.0/April 2010
- Current releases are GCC 4.5.3 and 4.6.0

GCC Portability

GCC is Ubiquitous

- All UNIXes, GNU/Linux, Solaris, Mac OS X, etc
- Windows via MinGW or Cygwin
- Cross-compiler for numerous mobile/embedded platforms

Plugin Support Portability

Plugins require support for dynamic loading

- Most UNIXes, including GNU/Linux, Solaris, and Mac OS X
- For Windows can be statically linked with some effort

Plugin Support Portability

Plugins require support for dynamic loading

- Most UNIXes, including GNU/Linux, Solaris, and Mac OS X
- For Windows can be statically linked with some effort

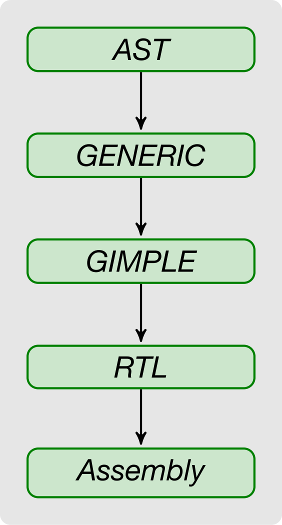
ODB works on the following platforms:

- GNU/Linux
- Windows
- Mac OS X
- Solaris

Plugin Implementation

- Plugin interface is a C API
- But plugin can be implemented in C++
- Or even in C++-0x
- Plugin is normally compiled with GCC

GCC Compilation Pipeline



Plugin Events

```
enum plugin_event
{
  PLUGIN_START_UNIT,      // Start translation unit.
  PLUGIN_FINISH_TYPE,     // Finished parsing a type.
  PLUGIN_PRE_GENERICIZE, // Finished parsing a function.
  ...                     // Lowering/optimization passes.
  PLUGIN_FINISH_UNIT      // Finished translation unit.
};
```

What a Plugin Can Do

- Source code generation
- Source code analysis
- Additional optimizations
- Instrumentations (source code)
- Instrumentations (object code)

What a Plugin Can Do

- Source code generation
- Source code analysis
- Additional optimizations
- Instrumentations (source code)
- Instrumentations (object code)

What a Plugin Can Do

- Source code generation
- Source code analysis
- Additional optimizations
- Instrumentations (source code)
- Instrumentations (object code)

What About the License

- GCC is GPLv3
- Plugin is GPL or more liberal (Boost, BSD, Apache)
- Generated source code can have any license

AST Introduction

- `global_namespace` is the root of the AST
- `tree` is an opaque *handle* for all the AST nodes
- Access to data stored in AST nodes is done via macros
- `TREE_CODE()` returns node's “type id”

AST Node Classes

- `*_DECL` nodes are declarations: `TYPE_DECL`, `VAR_DECL`
- `*_TYPE` nodes are types: `RECORD_TYPE`, `ARRAY_TYPE`
- Tree nodes can form linked lists
- `TREE_CHAIN()` can be used to traverse such lists

AST Node Classes

- `*_DECL` nodes are declarations: `TYPE_DECL`, `VAR_DECL`
- `*_TYPE` nodes are types: `RECORD_TYPE`, `ARRAY_TYPE`
- Tree nodes can form linked lists
- `TREE_CHAIN()` can be used to traverse such lists

```
tree decl = ...

for (; decl != 0; decl = TREE_CHAIN (decl))
{
  ...
}
```

Declaration Nodes

- A declaration names an entity in a scope
- `DECL_NAME()` returns declaration’s name
- For unnamed entities it returns `NULL`

Declaration Nodes

- A declaration names an entity in a scope
- `DECL_NAME()` returns declaration’s name
- For unnamed entities it returns `NULL`

```
tree decl = ...;
tree id (DECL_NAME (decl));
const char* name (IDENTIFIER_POINTER (id));
```

Declaration Nodes

- `TREE_TYPE()` returns type node
- `DECL_SOURCE_FILE()` returns source file
- `DECL_SOURCE_LINE()` returns source line

Namespace Declaration

- Namespace declaration node has `NAMESPACE_DECL` tree code
- Contains two lists: `names` and `namespaces`
- `global_namespace` is a `NAMESPACE_DECL` node

Namespace Declaration

- Namespace declaration node has `NAMESPACE_DECL` tree code
- Contains two lists: `names` and `namespaces`
- `global_namespace` is a `NAMESPACE_DECL` node

```
traverse_namespace (global_namespace);
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Nested Namespaces

```
void traverse_namespace (tree ns)
{
  traverse_declarations (NAMESPACE_LEVEL (ns)->names);

  for (tree decl = NAMESPACE_LEVEL (ns)->namespaces;
       decl != 0;
       decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

    print_declaration (decl);
    traverse_namespace (decl);
  }
}
```

Traversing Namespace Declarations

```
void traverse_declarations (tree decl)
{
  for (; decl != 0; decl = TREE_CHAIN (decl))
  {
    if (DECL_IS_BUILTIN (decl))
      continue;

     print_declaration (decl);
  }
}
```

Printing Declarations

```
void print_declaration (tree decl)
{
  int tc (TREE_CODE (decl));
  tree id (DECL_NAME (decl));
  const char* name (id
                    ? IDENTIFIER_POINTER (id)
                    : "<unnamed>");

  cerr << tree_code_name[tc] << " "
       << name << " at "
       << DECL_SOURCE_FILE (decl) << ":"
       << DECL_SOURCE_LINE (decl) << endl;
}
```

Printing Declarations

```
void print_declaration (tree decl)
{
  int tc (TREE_CODE (decl));
  tree id (DECL_NAME (decl));
  const char* name (id
                    ? IDENTIFIER_POINTER (id)
                    : "<unnamed>");

  cerr << tree_code_name[tc] << " "
       << name << " at "
       << DECL_SOURCE_FILE (decl) << ":"
       << DECL_SOURCE_LINE (decl) << endl;
}
```

Printing Declarations

```
void print_declaration (tree decl)
{
  int tc (TREE_CODE (decl));
  tree id (DECL_NAME (decl));
  const char* name (id
                    ? IDENTIFIER_POINTER (id)
                    : "<unnamed>");

  cerr << tree_code_name[tc] << " "
       << name << " at "
       << DECL_SOURCE_FILE (decl) << ":"
       << DECL_SOURCE_LINE (decl) << endl;
}
```

Printing Declarations

```
void print_declaration (tree decl)
{
  int tc (TREE_CODE (decl));
  tree id (DECL_NAME (decl));
  const char* name (id
                    ? IDENTIFIER_POINTER (id)
                    : "<unnamed>");

  cerr << tree_code_name[tc] << " "
       << name << " at "
       << DECL_SOURCE_FILE (decl) << ":"
       << DECL_SOURCE_LINE (decl) << endl;
}
```

Printing Declarations

```
void f ();

namespace n
{
  class c {};
}

typedef n::c t;
int v;
```

Printing Declarations

```
void f ();

namespace n
{
  class c {};
}

typedef n::c t;
int v;
```

```
var_decl         v at test.cxx:9
type_decl        t at test.cxx:8
function_decl    f at test.cxx:1
namespace_decl   n at test.cxx:4
type_decl        c at test.cxx:5
```

Declaration Order

```
var_decl       v at test.cxx:9
type_decl      t at test.cxx:8
function_decl  f at test.cxx:1
namespace_decl n at test.cxx:4
type_decl      c at test.cxx:5
```

Declaration Order

```
var_decl       v at test.cxx:9
type_decl      t at test.cxx:8
function_decl  f at test.cxx:1
namespace_decl n at test.cxx:4
type_decl      c at test.cxx:5
```

- Declaration order is often not preserved in AST
- In fact, some declarations are stored in different lists
- Plus all declarations for the same namespace are merged
- But we can restore order using source code line and column

Types

- *_TYPE tree codes
- `TREE_TYPE()` returns declaration's type

Type Categories

Fundamental Types

- VOID_TYPE
- BOOLEAN_TYPE
- INTEGER_TYPE

Type Categories

Derived Types

- `POINTER_TYPE`
- `REFERENCE_TYPE`
- `ARRAY_TYPE`

Type Categories

User-Defined Types

- `RECORD_TYPE`
- `UNION_TYPE`
- `ENUMERAL_TYPE`

CVR-Qualified Types

- Each type node contains a cvr-qualifier
- GCC makes a copy of a type to create a cvr-qualified version
- GCC can even have multiple copies of identical types
- `TYPE_MAIN_VARIANT()` returns *primary*, cvr-unqualified node

Class Types

- `RECORD_TYPE` nodes represent `class/struct` types
- In GCC AST types don't have names
- Instead types are *declared* to have names using `TYPE_DECL` nodes

Class Types

- `RECORD_TYPE` nodes represent `class/struct` types
- In GCC AST types don't have names
- Instead types are *declared* to have names using `TYPE_DECL` nodes

```
class c {...};
```

Class Types

- `RECORD_TYPE` nodes represent `class`/`struct` types
- In GCC AST types don't have names
- Instead types are *declared* to have names using `TYPE_DECL` nodes

```
class c {...};
```

As If

```
typedef class {...} c;
```

Class Types

```
class c {}; // AST: typedef class {...} c;
typedef c t;
```

Class Types

```
class c {}; // AST: typedef class {...} c;
typedef c t;
```

- `TYPE_DECL` nodes *imagined* by GCC are marked *artificial*
- This can be tested with the `DECL_ARTIFICIAL()` macro

Type Names

How do we get a type's name from the type node?

- `TYPE_NAME()` returns *a* type's `TYPE_DECL` node
- `TYPE_NAME (TYPE_MAIN_VARIAN())` returns artificial `TYPE_DECL`

Type Aliases

```
#ifdef _MSC_VER
typedef __int64 myint;
#else
typedef long long myint;
#endif

myint i;
```

```
tree vd = ... // i's declaration node (VAR_DECL)
tree t = TREE_TYPE (vd);
tree d1 = TYPE_NAME (t); // myint
tree d2 = TYPE_NAME (TYPE_MAIN_VARIANT (t)); // long long
```

Type Aliases

```
#ifdef _MSC_VER
typedef __int64 myint;
#else
typedef long long myint;
#endif

myint i;
```

```
tree vd = ... // i's declaration node (VAR_DECL)
tree t = TREE_TYPE (vd);
tree d1 = TYPE_NAME (t); // myint
tree d2 = TYPE_NAME (TYPE_MAIN_VARIANT (t)); // long long
```

Type Aliases

```
#ifdef _MSC_VER
typedef __int64 myint;
#else
typedef long long myint;
#endif

myint i;
```

```
tree vd = ... // i's declaration node (VAR_DECL)
tree t = TREE_TYPE (vd);
tree d1 = TYPE_NAME (t); // myint
tree d2 = TYPE_NAME (TYPE_MAIN_VARIANT (t)); // long long
```

Type Aliases

```
#ifdef _MSC_VER
typedef __int64 myint;
#else
typedef long long myint;
#endif

myint i;
```

```
tree vd = ... // i’s declaration node (VAR_DECL)
tree t = TREE_TYPE (vd);
tree d1 = TYPE_NAME (t); // myint
tree d2 = TYPE_NAME (TYPE_MAIN_VARIANT (t)); // long long
```

Type Aliases

```
#ifdef _MSC_VER
typedef __int64 myint;
#else
typedef long long myint;
#endif

myint i;
```

```
tree vd = ... // i’s declaration node (VAR_DECL)
tree t = TREE_TYPE (vd);
tree d1 = TYPE_NAME (t); // myint
tree d2 = TYPE_NAME (TYPE_MAIN_VARIANT (t)); // long long
```

Class Information

- `TYPE_BINFO()` returns the base class vector
- `TYPE_FIELDS()` returns the list of member variables and nested type declarations
- `TYPE_METHODS()` returns the list of member functions

Class Information

```
class b1 {};
class b2 {};
class c: protected b1,
         public virtual b2
{
  int i;
  static int s;
  void f ();
  c (int);
  ~c ();
  typedef int t;
  class n {};
};
```

Class Information

```
class b1 at test.cxx:1
class b2 at test.cxx:2
class c at test.cxx:5
        protected base b1
        public virtual base b2
field_decl    c::i type integer_type at test.cxx:6
var_decl      c::s type integer_type at test.cxx:7
function_decl c::f type method_type at test.cxx:8
function_decl c::c type method_type at test.cxx:9
function_decl c::c type method_type at test.cxx:10
type_decl     c::t type integer_type at test.cxx:11
class         c::n at test.cxx:12
```

#include Hierarchy

```
#include <cstddef>
```

#include Hierarchy

```
#include cstddef line 1
  #include c++config.h line 43
    #include os_defines.h line 392
      #include /usr/include/features.h line 40
        #include /usr/include/bits/predefs.h line 313
        #include /usr/include/sys/cdefs.h line 346
         #include /usr/include/bits/wordsize.h line 353
        #include /usr/include/gnu/stubs.h line 378
          #include /usr/include/bits/wordsize.h line 4
          #include /usr/include/gnu/stubs-64.h line 9
    #include cpu_defines.h line 395
  #include stddef.h line 44
```

Things We Haven't Covered

- Function declarations
- Function bodies
- Templates

AST Summary

- Ugly “we can do polymorphism in C” interface
- While some things might be inconvenient
- There is usually a way to get the information you need

Pragmas and Attributes

```
#pragma version(1.2)
struct coord
{
  float lat __attribute__ ((doc ("Latitude.")));
  float lon __attribute__ ((doc ("Longitude.")));
};
```

Pragmas and Attributes

```
#pragma version(1.2)
struct coord
{
  float lat __attribute__ ((doc ("Latitude.")));
  float lon __attribute__ ((doc ("Longitude.")));
};
```

- Traditionally ad-hoc C++ “preprocessors” use comments
- Plugin can register custom pragmas and attributes
- Thus we can have a DSL embedded in C++
- OpenMP is a well-known example of such a DSL

Pragmas

Register pragmas during the `PLUGIN_PRAGMAS` event.

```
c_register_pragma ("", "version", handle_version);
```

Pragmas

Register pragmas during the `PLUGIN_PRAGMAS` event.

```
c_register_pragma ("", "version", handle_version);
```

```
extern "C" void handle_version (cpp_reader* reader)
{
  tree t;
  cpp_ttype tt = pragma_lex (&t);
  if (tt == CPP_OPEN_PAREN)
  {
  ...
}
```

Pragmas

Register pragmas during the `PLUGIN_PRAGMAS` event.

```
c_register_pragma ("", "version", handle_version);
```

```
extern "C" void handle_version (cpp_reader* reader)
{
  tree t;
  cpp_ttype tt = pragma_lex (&t);
  if (tt == CPP_OPEN_PAREN)
  {
  ...
}
```

Pragma handlers can access AST (lookup types, etc).

Attributes

Register attributes during the `PLUGIN_ATTRIBUTES` event.

```
static struct attribute_spec attr =
 {"doc", 1, 1, false,  false, false, handle_doc, false};

register_attribute (&attr);
```

Attributes

Register attributes during the `PLUGIN_ATTRIBUTES` event.

```
static struct attribute_spec attr =
 {"doc", 1, 1, false,  false, false, handle_doc, false};

register_attribute (&attr);


extern "C" tree handle_doc (tree node,
                            tree name,
                            tree args,
                            int flags,
                            bool* dont_add)
{
  return NULL_TREE;
}
```

Attributes

- Attributes are added to tree nodes
- `DECL_ATTRIBUTES()` returns list of attributes for a declaration
- `TYPE_ATTRIBUTES()` returns list of attributes for a type

Runtime Template Instantiation

```
class c
{
  int i;                 // -> row
  std::vector<int> v; // -> table
};
```

Runtime Template Instantiation

```
class c
{
  int i;                  // -> row
  std::vector<int> v; // -> table
};

if (name == ”::std::vector”)
```

Runtime Template Instantiation

```
class c
{
  int i;                 // -> row
  std::vector<int> v; // -> table
};

if (name == "::std::vector")

if (name == "::std::vector" ||
    name == "::boost::unordered_set")
```

Runtime Template Instantiation

```
class c
{
  int i;              // -> row
  std::vector<int> v; // -> table
};

if (name == "::std::vector")

if (name == "::std::vector" ||
    name == "::boost::unordered_set")

if (name == "::std::vector" ||
    name == "::boost::unordered_set" ||
    custom_containers.find (name))

g++ -fplugin-arg-odb-container=::google::sparse_hash_set ..
```

Runtime Template Instantiation

Things get worse:

```
class c
{
  std::vector<int> v;   // CREATE TABLE (index, value)
  std::set<int> s;      // CREATE TABLE (value)
  std::map<int, int> m; // CREATE TABLE (key, value)
};
```

Runtime Template Instantiation

Q: How would we handle this in C++?

Runtime Template Instantiation

Q: How would we handle this in C++?

A: We would use traits.

Runtime Template Instantiation

Q: How would we handle this in C++?

A: We would use traits.

```
enum container_kind {ck_ordered, ck_set, ck_map};

template <class T>
struct container_traits;

template <class T>
struct container_traits<std::vector<T> >
{
  static const container_kind kind = cl_ordered;
};
```

Runtime Template Instantiation

- A plugin can instantiate a template *at runtime*
- And then examine the resulting instantiation

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name ("container_traits");

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, "kind");
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name ("container_traits");

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, "kind");
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name ("container_traits");

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, "kind");
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name (”container_traits”);

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, ”kind”);
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name ("container_traits");

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, "kind");
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Runtime Template Instantiation

```
tree type = ... // Type we would like to test.
tree traits = lookup_qualified_name ("container_traits");

tree args = make_tree_vec (1);
TREE_VEC_ELT (args, 0) = type;

tree inst = lookup_template_class (traits, args);
inst = instantiate_class_template (inst);

if (inst != error_mark_node && COMPLETE_TYPE_P (inst))
{
  tree kind = lookup_qualified_name (inst, "kind");
  // Extract initial value from VAR_DECL.
}
else
  // Not a container.
```

Compilation Prologue/Epilogue

- Traits approach is elegant
- But how do we include traits into the translation unit?

Compilation Prologue/Epilogue

- Traits approach is elegant
- But how do we include traits into the translation unit?

```
#include <std-container-traits.hxx>
#include <boost-container-traits.hxx>
#include <google-container-traits.hxx>

class c
{
  std::vector<int> v;
  boost::unordered_set<int> s;
  google::sparse_hash_map<const char*, int> m;
};
```

Compilation Prologue/Epilogue

This doesn’t work:

```
g++ *-container-traits.hxx test.cxx
```

Compilation Prologue/Epilogue

- We want to add code before/after the file being compiled
- While maintaining original file/line information
- This is tricky but possible

Compilation Prologue/Epilogue

- GCC can parse `STDIN`
- We are going to “pipe” synthesized translation unit to GCC
- Using the `#line` preprocessor directives for file/line illusion
- Plugin will set the main file directory for `#include ""`

Compilation Prologue/Epilogue

Synthesized translation unit:

```
#line 1 ”<prologue>”
#include <std-container-traits.hxx>
#include <boost-container-traits.hxx>
#include <google-container-traits.hxx>

#line 1 ”test.cxx”
class c
{
  std::vector<int> v;
  boost::unordered_set<int> s;
  google::sparse_hash_map<const char*, int> m;
};

#line 1 ”<epilogue>”
```

GCC vs Clang (Subjective)

- Non-existent documentation or examples
- AST borrows from GCC
- "AST is essentially immutable"
- No support for custom pragmas/attributes

GCC vs Clang (Subjective)

- Better diagnostics
- More lexical information preserved in AST
- Support for source code rewriting

GCC vs Clang (Subjective)

- Plugin interface — undocumented and “not very robust”
- Create your own driver — large amount of boilerplate code
- Hack Clang source code directly — preferred method

GCC vs Clang (Very Subjective)

- BSD licensed and developed by a single company (Apple)
- Iffy support for platforms other than Mac OS X and Linux
- Delayed distribution packaging, no `clang-dev`
- Doesn’t feel like production-quality yet

GCC vs Clang (Very Subjective)

- BSD licensed and developed by a single company (Apple)
- Iffy support for platforms other than Mac OS X and Linux
- Delayed distribution packaging, no `clang-dev`
- Doesn't feel like production-quality yet

```
clang -cc1 -boostcon test.cxx
```

GCC vs Clang (Very Subjective)

- BSD licensed and developed by a single company (Apple)
- Iffy support for platforms other than Mac OS X and Linux
- Delayed distribution packaging, no `clang-dev`
- Doesn't feel like production-quality yet

```
clang -cc1 -boostcon test.cxx
```

"This structure is somewhat mystical, but after meditating on it, it will make sense to you :)."

GCC Plugin Applications

What applications can we build with GCC Plugins?

Code Generation

Code Generation

- Documentation/class diagram generation
- Dynamic object introspection/extended RTTI
- Object persistence (XML, RDBMS, etc)
- Remote invocation (RPC)
- Binding for other languages
- Transactional objects
- Embedded DSL (command line parsing, config formats, GUI)

Source Code Analysis

Source Code Analysis

- C++ code browser
- Checking of naming conventions
- Reuse analysis
- Statistical analysis

Source Code Instrumentation and Rewriting

Source Code Instrumentation and Rewriting

- Automatic locking
- Data access monitoring
- Execution tracing (DTrace)

Other

Other

- Semantic Graph for C++
- AST serialization

Summary

We now have a C++ parser that is

- open-source,
- cross-platform,
- widely-deployed, and
- mature.

Resources

- Parsing C++ with GCC plugins, Part 1, Part 2, and Part 3
  - http://www.codesynthesis.com/~boris/blog/
- GCC Internals Documentation
  - http://gcc.gnu.org/onlinedocs/gccint/
- GCC Wiki Plugin page
  - http://gcc.gnu.org/wiki/plugins
- GCC source code
- GCC test suite

Questions

?