metacode and Compile-Time Reflection

ACCU Conference 2006
Detlef Vollmann

metacode and Compile-Time Reflection

Detlef Vollmann
vollmann engineering gmbh
Lucerne, Switzerland

eMail: dv@vollmann.ch
http://www.vollmann.ch/

Overview

metacode

DB Example

Threading Example

AOP

Discussion

metacode

- metacode presentation by Daveed Vandevoorde at ACCU conference 2003
  - no C++ proposal yet
  - not much publicly visible work since 2003
- metacode mechanisms
  - compile-time functions
  - code injection
- metacode library
  - namespace `stdmeta`
  - `type`, `id`, `string_literal`, `array`, `table`
  - `is_accessible()`, `is_lvalue()`, `in_normal_function()`, `error()`

Function Example

[from original presentation]

```
template <typename T> metacode
T power(T b, unsigned n) {
  T r = 1;
  for (int k = 0; k<n; ++k) t *= b;
  return r;
}

float a1[power(2, 3)];    // array size compile-time evaluated

int p = 3;
float a2[power(2, p)];   // error: value of p not available at
                         // compile-time
```


Function Example Insights

- metacode functions
- ... can be templates
  - can also be non-templates
- ... can have normal parameters
  - but no implicit conversion
- ... can use normal declarations, control structures and expressions
- From a programmer's view, metacode functions are just normal functions that cannot access runtime values

Injection Example 1

[from original presentation]

```
metacode
double mypow(double b, int n) {
  using ::stdmeta::is_constant;
  if (is_constant(b) &&
      is_constant(n) &&
      n >= 0) {
    return power<>(b, (unsigned)n);
  } else {
    return-> ::std::pow(b, n);
  }
}
```


Injection Example 1 Insights

- special kind of "magic" type for built-in functions
  - `is_constant(xyz)`
- no implicit conversion
  - but explicit casts
- metacode functions can return normal values
  - as seen before
- ... or can return code to be injected

Injection Example 2

[from original presentation]

```
metacode define_fields(array<type> types) {
  for (int k; k<types.length(); ++k) {
    type FieldT = types[k];
    id FieldName = id("field" +
                      string_literal(k));
    metacode-> {
      FieldT FieldName;
    }
  }
}
```


Injection Example 2 Insights

- no return type
  - actually makes sense: it is not void, but it is also no other type
- container template types
  - sequential `array<>` (possibly not the same as `std::tr1::array`)
  - associative `table<>`
- `stdmeta::type`
  - base for reflection
- `stdmeta::id`
  - useful for injection
- `stdmeta::string_literal`
  - string manipulation for identifiers (only?)

Overview

metacode

DB Example

Threading Example

AOP

Discussion

DB Example

[this is probably along the lines of Daveed Vandevoorde's ideas]

- DB Libraries are classic examples for reflection
  - persistence in general
- General approach:
  - iterate through data members
    - recursively
  - private and public
  - save them type specific
    - use member name as field name

Usage

```
// DB example

class X
{
    // ...
private:
    int i;
    double d;
    string s;
    pair<int, int> p;
};

void foo()
{
    X myX;
    DbSaveNewRecord(myX, myDb, "MyTable");
}
```


Implementation

```
metacode DbSaveNewRecord(id obj, id db, string_literal tabName)
{
    type t(obj);
    if (!is_aggregate(t))
    {
        error("only aggregate types can be stored as new records, but"
                + string_literal(obj) + "isn't.");
    }
    else
    {
        metacode->
        {
            DbRecord rec(db.prepareNewRecord(tabName));
            DbSaveMember<t.the_type()>(rec, obj, "");
            rec.insert();
        }
    }
}
```

Implementation

```
template <typename T> metacode DbSaveMember(id rec, id obj,
                                            string_literal memberName)
{
    // just delegate to general implementation
    DbSaveGeneral(T, rec, obj, memberName);
}
template <> metacode DbSaveMember<string>(id rec, id obj,
                                          string_literal memberName)
{
    if (memberName.empty()) {
        error("strings can only be stored as members of aggregates");
    } else {
        metacode-> {
            {
               string dbTypeName(rec.getTypeName(memberName));
               DbSaver *saver(lookupSaverObject(rec, "string", dbTypeName));
               saver->save(rec, memberName, obj.id(memberName));
}   }   }   }

// alternative approach
template <> metacode DbSaveMember<string>(id rec, id obj, id member)
{    // ...
                saver->save(rec, string_literal(member), obj.member);
}
```


Implementation

```
metacode
DbSaveGeneral(type t, id rec, id obj, string_literal memberName)
{
    if (is_aggregate(t.the_type()))
    {
        type::iterator memberIt(t.data_member_begin());
        while (memberIt != t.data_member_end())
        {
            metacode-> DbSaveMember<memberIt->type().the_type()>
                           (rec, obj, memberName + "." + memberIt->name());
        }
    }
    else
    {
        metacode->
        {
            {
                string dbTypeName(rec.getTypeName(memberName));
                DbSaver *saver(lookupSaverObject(rec, "int", dbTypeName));
                saver->save(rec, memberName, obj.id(memberName));
            }
        }
    }
}
```

Details / Issues

- `DbSaveNewRecord(myX, myDb, "MyTable");`
  `void DbSaveNewRecord(id obj, ...`
  - implicit conversion from (any named) object to `id`
- `type t(obj);`
  - Is this possible?
  - Lookup rules for `id`
  - Additional type `object`?
- `DbSaveMember<t.the_type()>(rec, obj, "");`
  - We need a way to get the type for template instantiation.
    - Or else it is not possible to instantiate a template with `type` or overload a function on `type`, which is also an option.

Details / Issues

- `template <> metacode void DbSaveMember<string>`
  - metacode template specialization
- `DbSaveGeneral(T, rec, obj, memberName);`
  - implicit conversion from any ("normal") type to `type`
- `saver->save(rec, memberName, obj.id(memberName));`
  - `obj.id(memberName)` injects the correct object member access expression `myX.i`

Resulting Code

```
void foo()
{
    X myX;
    //DbSaveNewRecord(myX, myDb, "MyTable");
    DbRecord rec(db.prepareNewRecord(tabName));
    // DbSaveMember<X>(rec, myX, "");
    //  DbSaveGeneral(X, rec, myX, "");
    //   DbSaveMember<int>(rec, myX, ".i");
    //    DbSaveGeneral(int, rec, myX, ".i");
    {
        string dbTypeName(rec.getTypeName(".i"));
        DbSaver *saver(lookupSaverObject(rec, "int", dbTypeName));
        saver->save(rec, ".i", myX.i);
    }
    //   DbSaveMember<double>(rec, myX, ".d");
    //    DbSaveGeneral(double, rec, myX, ".d");
    {
        string dbTypeName(rec.getTypeName(".d"));
        DbSaver *saver(lookupSaverObject(rec, "double", dbTypeName));
        saver->save(rec, ".d", myX.d);
    }
```


Resulting Code

```
    //   DbSaveMember<string>(rec, myX, ".s");
    {
        string dbTypeName(rec.getTypeName(".s"));
        DbSaver *saver(lookupSaverObject(rec, "string", dbTypeName));
        saver->save(rec, ".s", myX.s);
    }
    //   DbSaveMember<pair<int, int> >(rec, myX, ".p");
    //    DbSaveGeneral(pair<int, int>, rec, myX, ".p");
    //     DbSaveMember<int>(rec, myX, ".p.first");
    //      DbSaveGeneral(int, rec, myX, ".p.first");
    {
        string dbTypeName(rec.getTypeName(".p.first"));
        DbSaver *saver(lookupSaverObject(rec, "int", dbTypeName));
        saver->save(rec, ".p.first", myX.d);
    }
    //     DbSaveMember<int>(rec, myX, ".p.second");
    //      DbSaveGeneral(int, rec, myX, ".p.second");
    {
        string dbTypeName(rec.getTypeName(".p.second"));
        DbSaver *saver(lookupSaverObject(rec, "int", dbTypeName));
        saver->save(rec, ".p.second", myX.d);
    }
    rec.insert();
}
```

Overview

metacode

DB Example

Threading Example

AOP

Discussion

Threading Example Usage

[this is not along the lines of Daveed Vandevoorde's ideas, as it exposes the AST]

- Populating an array in parallel:

```
double roots[10];

void populate_roots()
{
    parallel_for<int>(i, 0, 10, 1)
    {
        d[i] = sqrt(i);
    }
}
```

Implementation

```
metacode
template <typename T>
parallel_for(id var, T start, T end, T step, block b)
{
    static int funcCounter = 0;

    ++funcCounter;
    // injecting an empty function definition for thread
    id funcName("f" + string_literal(funcCounter);
    metacode->ThreadImplNamespace
    {
        void funcName(id(T) var) {}
    }
    // injecting the function body
    for (code::statement_iterator s(b.statement_begin());
         s != b.statement_end();
         ++s)
    {
        metacode->ThreadImplNamespace::funcName
        {
            s;  // maintain order
        }
    }
```


Implementation

```
    // injecting thread management
    metacode->
    {
        {
            thread_group workers;
            for (T j = start; j != end; j += step)
            {
                workers.create_thread(ThreadImplNamespace::funcName, j);
            }

            workers.join_all();
        }
    }
}
```

Details / Issues

- `parallel_for(id var, T start, T end, T step, block b)`
  - `block b` matches the block after the closing parenthesis
  - might cause problems with ellipsis (`...`)
- static for compile-time functions
  - `code` type
- injecting full function definitions seperately
  - first empty body, then inject into that body
  - keep order
- iterate through statement sequence

Resulting Code

```
// d[i] = sqrt(i);
void ThreadImplNamespace::f1(int i)
{
    d[i] = sqrt(i);
}
void populate_roots()
{
    // parallel_for<int>(i, 0, 10, 1)
    {
        thread_group workers;
        for (int i = 0; i != 10; i += 1)
        {
            workers.create_thread(ThreadImplNamespace::f1, i);
        }
        workers.join_all();
    }
}
```

Overview

metacode

DB Example

Threading Example

AOP

Discussion

AOP

- adding a tracing aspect to a function:

```
void goo();
add_trace(goo);

void goo()
{
    // ...
}
```

- what we want to get:

```
void goo()
{
    clog << "Entering goo..." << endl;
    // ...
    clog << "Leaving goo..." << endl;
}
```

- same approach can be used to check pre / post conditions
  - can be done at call site or in function
    - but optimization benefits only when in function

Implementation

```
metacode
add_trace(id f)
{
    block b(f.get_function_body());

    // inject at begin
    code::statement_iterator s(b.statement_begin());
    ++s;
    metacode->s
    {
        clog << string_literal("Entering " + string_literal(f) + "...") << endl;
    }

    // inject at end
    s = b.statement_end();
    metacode->s
    {
        clog << string_literal("Leaving " + string_literal(f) + "..." << endl;
    }
}
```


Details / Issues

- inject at specific places
  - possibly first search for the place, then inject

```
for (code::statement_iterator s(b.statement_begin());
     // ...
  if (check_for_some_property(*s))
    metacode->s { ... } // insert before s
```

  - injecting code into an already closed scope is problematic!
    - is the definition available?
    - ODR
- a different option would be to add changes at the call sites
  - also viable for pre / post conditions
    - but optimization benefits only when in function

Overview

metacode

DB Example

Threading Example

AOP

Discussion

Discussion

- some tasks effectively require reflection
- most runtime reflection can (also) be done with compile-time reflection
- some features that are normally part of a language can be implemented by a library

Disclaimer

- all this is highly speculative
  - no implementation yet
    - some parts are implemented by Daveed Vandevoorde in an internal version of the EDG front end
- possibly never in ISO C++
  - quite probably not in C++0x
  - but possibly in a TR
  - probably with lots of changes

References

- Daveed Vandevorde's original paper:
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1471.pdf
- Discussion about metacode:
  - http://www.vandevoorde.com/Daveed/News/Archives/000015.html
- possibly updated version of this paper at
  - http://www.vollmann.ch/en/pubs/accu06meta.html

