Document Number: N3635
Date: 2013-04-29
Authors: Ian McIntosh
Yaoqing Gao
Christophe Cambly
Chris Bowler
Hubert Tong
Chris Lord
Sean Perry
Raymond Mak
Raul Silvera
Michael Wong, IBM, michaelw@ca.ibm.com
Project: Programming Language C++, EWG
Reply to: michaelw@ca.ibm.com
Revision: 1

Towards restrict-like semantics for C++

Abstract

This paper is a proposal to enhance N3538 in the pre-Bristol mailing with feedback, and summarizes reflector discussion [4] on restrict–like semantics. It analyzes N3538 and suggests improvements on it as well as additional solutions. It is a work in progress to be proposed for discussion at Bristol for N3538. As such, it should be read with caution as it may be incomplet, incorrek, and inconssitent. It should be read as a point of proposed collaboration with the authors of N3538, Lawrence Crowl from Google, and other parties who have already indentified their interest including Clark Nelson of Intel, Darryl Gove [14] of Oracle and anyone else interested in collaborating on improving the performance of C++ through improved aliasing.

1. Problem Domain

N3538 [1] makes an argument for efficiency and expressiveness for C++ and finds that they conflict with each other. We agree. However, there is no question that restrict qualifier benefits compiler optimization in one way or another and allows improved code motion and elimination of loads and stores. Since the introduction of C99 restrict, it has been implemented for C++ with several compiler implementations. But the feature is brittle in C++ without clear rules for C++ syntax and semantics. Now with the introduction of C++11, functors are being replaced with lambdas and users have begun asking how to use restrict in the presence of lambdas. There is no official restrict support in C++. But the fact that it is supported by many compilers, means that it is a problem that should be solved before its use becomes dominant and wild when combined with C++11 constructs. Some possible issues with restrict in C++ are:

- Restrict Class members and indirection with “this pointer”
- Passing of restrict qualifiers into functions, functors, lambdas, and templates
- Escaping of restrict pointer values inside functions
- Overlapping array members, strides

The primary need is not about ensuring that the programmer can write code that is robust in the presence of aliasing, but to provide a mechanism by which the programmer can communicate to the compiler the absence of aliasing, which will be enforced by the programmer through his own means.

2. Compiler implementations and related work

Some initial internet research indicate restrict keyword (or its variants with underscores) is accepted by

- Microsoft[5]
- GCC [6]
- IBM [7]
- Intel (EDG) [8]
- HP(EDG) [9]
- Solaris [10]

Solaris seems to only accept it as a command line option that modifies all pointers in a file. IBM's #pragma disjoint [11] modifies dereferences of specific pointers. This indicates there is wide-spread demand for restrict-semantics, or at least an appetite for the performance benefit offered by restrict. Users typically have to jump through considerable hoops to get its effect through code rewriting through temporaries, or factoring and inlining function bodies to simulate its effect.

Such cases often indicate a need for language support.

Aliasing has been also a problematic area in C and C++, since C++ essentially adapts its aliasing rules from C in Clause 6.5p7. Several papers by Clark Nelson has attempted to fix the aliasing rules [2][3]. Although the discussion in these papers on effective types is interesting, we believe it is orthogonal to our discussion.

3. Issues with restrict with C++

C99 restrict qualifier support is not part of C++11. At the time of reviewing C99 feature inclusion in C++ during the Mont Tremblant meeting, restrict was considered but was waiting a paper proposal although none came forward. C99 restrict even has ambiguities in C as it unclear whether the phrase covers struct members [13].

C99 restrict syntax in C++ have even more issues to resolve. Restrict is a C99 feature and was never designed to work in class abstractions and it may have to do with that pointers are not common in C++. In essence, it was designed for fine-grain aliasing for C, but not well-designed for type-based aliasing in C++.

We need to resolve how the qualifier will be carried through functors and templates. We need to determine if the qualifier changes the overload set.

```
void foo(int * __restrict__ * a) { printf("First\n"); } // candidate function
1.
void foo(int * * b) { printf("Second\n"); }            // candidate function
2.

int main() {
   int a;
   int *b=&a;
   int *__restrict__ *c=&b;
```

```
   int * *d =&b;
   foo(c);
   foo(d);
}
```

Does it participate in mangling of the type? Does it affect type specifications in the same way as a cv-qualifier?

```
int * __restrict__ p;      // p is a restricted pointer to int
int * __restrict__ * q;    // q is a pointer to a restricted pointer to int
void foo(float * __restrict__ a, float * __restrict__ b);
                           // a and b point to different (non-overlapping)
objects
```

At its core, this seems to be a question on how the C++ standard defines aliasing rules for these situations because the restrict scope is not explicitly present in the operator() overload of the functor (or lambdas for C++11).

However, C++ should have a better way to rewrite the following code to enable it to be optimized (and possibly SIMDized)

```
typedef double * __restrict__ d_p ;


class Functor
{
public:
  Functor(d_p x, d_p y, d_p z, d_p u,
          double q, double r, double t)
  : m_x(x), m_y(y), m_z(z), m_u(u), m_q(q), m_r(r), m_t(t) { ; }

  void operator() (int k)
  {
    m_x[k] = m_u[k] + m_r*( m_z[k] + m_r*m_y[k] ) +
          m_t*( m_u[k+3] + m_r*( m_u[k+2] + m_r*m_u[k+1] ) +
          m_t*( m_u[k+6] + m_q*( m_u[k+5] + m_q*m_u[k+4] ) ) );
  }
  private:
  d_p m_x;
  const d_p m_y;
  const d_p m_z;
  const d_p m_u;
  const double m_q;
  const double m_r;
  const double m_t;
};
void functor(d_p x, d_p y, d_p z, d_p u,
       double q, double r, double t,
       int length)
{
  Functor kernel(x, y, z, u, q, r, t);

  for (int k=0 ; k<length; ++k) { //1
    kernel(k) ;
```

```
  }
}
```

Users want to specify restricted pointers for the loop //1. The member initializations indicate all pointers are restricted, which are intended for the loop //1. The problem is that the call operator, which uses the restrict pointer members can make little use of the restrict property. Restrict is a promise of an exclusive handle to memory at time of initialization, but does not guarantee that the value does not escape after initialization. The call operator must conservatively assume the restrict pointer values have escaped prior to entry and indirect operations through these pointers must still alias indirect operations through non-restrict pointers of appropriate type.

Although the restrict pointer values may escape, it can be safely assumed that m_x, m_y, m_z and m_u remain disjoint, which should be sufficient to get the desired optimization in this case. If that is the user's intention, the use of restrict here is a clever but obfuscated means of expressing disjoint properties of the member pointers.

Inlining may relieve the problem if enough inlining is performed so that the performance sensitive code is inlined into the scope of the owning object and it can be proven the restrict pointer values do not escape after initialization. The fundamental problem, however, of the pointer value possibly escaping and hindering optimization remains and may be quite problematic in cases where restrict pointer members are referenced indirectly (including through the implicit-this pointer.) In the case of member declarations, it's more useful to the compiler if the restrict promise is an exclusive handle to memory for the lifetime of the owning object.

For the above example, the member function needs to be processed independently. It is possible the restrict pointer member values have escaped prior to calling the function. While we may be able to take advantage of the fact that restrict pointers that are members of the same object are always disjoint, the "implicit-this" indirection of accessing the pointers is still problematic because it's not safe to assume that restrict pointers in different objects are disjoint, especially after optimization.

With respect to lambdas, the same logic applies. Capture of a restrict pointer is not defined by the C++11 Standard or any standard, but it's reasonable to expect that the capture of restrict pointers preserves restrictness. We should be able to make the restrict indirects in the lambda function disjoint from each other. This is another benefit of specifying the properties of restrict-like semantics for C++.

The usual solution of the above example is enlisted partially by N3538, using extra copies of temporaries. In addition, users can also wrap each one of these into distinct object types to say there is no aliasing. But this adds a further level of indirection to the restrict symbol. In general, C99 restrict in a class abstraction is not well defined. However, one of our proposed solutions does take advantage of this and tries to minimize the indirection issue.

Another solution is simply to remove the functor abstraction, and simply rewrite the code inlined:

```
void functor(d_p x, d_p y, d_p z, d_p u, double q, double r, double t, int length)
{
  for (Indx_type k=0 ; k<length; ++k) {
    x[k] = u[k] + r*( z[k] + t*y[k] ) +
    t*( u[k+3] + t*( u[k+2] + t*u[k+1] ) +
    t*( u[k+6] + q*( u[k+5] + q*u[k+4] ) ) );
  }
}
```

This is impractical for most large scale code deployment and seriously breaks the composition ability of C++.

Users would like to exploit C++11 features such as lambda expressions. They explicitly express their requirements that they will use C++11 only if it can deliver better performance. Here is a quote from one user:
“They are willing to write compile-friendly code, specially, they are willing to add "restrict keyword" to help the compiler generate better code.”. The above example shows a case where the pointers in the loop body are currently "not" restricted because it is potentially not carried to functors by all compilers. Similar cases can be made for templates.

Users are now experimenting with the ability to decouple a loop body from a loop traversal in a systematic way that may enable them to achieve a high-level of performance portability across diverse future platforms.

They can define a template method that defines a traversal over an arbitrary loop body and a “tag struct” used to instantiate the traversal method.

```
struct vectorized_traversal {};

template <typename LOOP_BODY>
inline void IndexSet_forall(vectorized_traversal,
int begin, int end,
LOOP_BODY loop_body)
{
  for ( int ii = begin ; ii < end ; ++ii ) {
    loop_body( ii );
  }
}
```

Then, the earlier examples would look like the following:

```
void functor_template(d_p x, d_p y, d_p z, d_p u, double q, double r, double t, int length)
{
  Functor kernel(x, y, z, u, q, r, t);
  IndexSet_forall<vectorized_traversal>(0, length, kernel);
}
```

This example also motivates the desire to have compiler support for C++ lambda expressions. They are much less cumbersome to write and use, and thus more desirable, than C++ function objects.

Enabling better performance has wider implications. The above loop, when properly aliased, can be vectorized automatically without explicit directives. As we move to enable full access of the performance of a machine, this becomes much more important.

4. Feedback for N3538

N3538 proposes several solutions including the general advice by many C++ gurus of passing by value (too expensive due to temporary construction), passing by (const) reference, which introduces aliasing implications. One can overcome the aliasing implications with users adding additional direct copies to temporaries which can be cumbersome.

It enlists three solutions to reduce the problem with using const reference and discusses their merits:

- pass small classes by value
  - avoids aliasing issues
  - remove some indirect references when accessing parameters
  - may introduce copying on older platforms
  - net performance gain unclear
- adapt restrict qualifier for C++
  - burden is on the programmer to avoid aliasing
  - does not take full advantage of modern calling convention
- design a new language feature
  - Carefully targeted to make code less sensitive to the platform and enable compilers to better optimize the program

We fully endorse this approach. We feel something like the second solution, but fully specifying restrict-like semantics should be pursued or the third solution should be discussed with the goal of designing a new language feature for making code less sensitive to the platform and enable compilers to better optimize program, exactly as stated in the paper.

However, we have feedbacks concerning N3538 which we hope to incorporate in collaboration with the author in future papers. In particular, we feel the solutions do not go far enough in some ways. Runtime versioning mechanisms as described in the paper are not practical in the presence of multiple parameters and complex access patterns or partial overlaps (e.g. striding). These complex cases exist in the field very pervasively.

Here are several stride examples that are common in the field:

a. The definition of this is using two pointers to walk an array and the pointers never point to the same thing.

   Example #1

```
void dupElems(int* t, int *s, int n) {
  for (int i=0; i<n; ++i, s+=2, t+=2) {
    *t=*s;
  }
}
dupElems(arr, arr+1, sizeof(arr)/2);  // copies odd elements to the preceding even element
```

b. Another example that is better known is

```
void strcpy(char *t, char *s) {
  while (*t++=*s++);
}
```

   Matrix operations can benefit greatly from proper handling of striping.

   In both of these examples the stores should not cause the optimizer to consider the source modified. I suspect these are motivating examples for the restrict qualifier as adding restrict to the parameter declarations provides enough information to the optimizer to do the right thing.

c. Here is another example where restrict doesn't solve the problem without extra work:

```
String::String(const char *str, unsigned len)
  {
```

```
_curLen = len;
if (len <= localBufferSize)
   {
   _maxLen = localBufferSize;
   _str =  _buffer;
   }
else
   {
   for (_maxLen = remoteBufferInitSize;
        _maxLen <= len; _maxLen += (unsigned int)remoteBufferIncrSize)
      {}
   _str = new StringChar[_maxLen+1];
   }
for (int i = 0; i < len; i++)
  _str[i] = str[i];
_str[len] = '\0';
}
```

The loop to copy str into _str could be faster if the compiler back end knew the two pointers were disjoint. Before the loop we could assign _str to a restrict pointer but the user shouldn't need to add an extra local variable to get the right optimizations done.

Most of the solutions in N3538 focus on restricting function arguments. It seems to ignore members, overlapping arrays, unions, and does not cover C++11 lambda arguments or captures. A mechanism such as the "EitherOr" being provided would be useful if it allows the programmer to communicate the absence of aliasing to the compiler. But for that to work the semantics need to be different. It would mean "have the same semantics as pass-by-value as long as the parameters don't overlap, otherwise it has undefined semantics". It would also require these parameters not to overlap with any other objects accessed directly in the function scope. This would require some specification work to define precisely what "overlaps" means.

Such a mechanism would also be required at scopes other than function parameters. Effectively it would require this new "either-or reference" to be placed anywhere a reference can exist.

The solutions do support 3 points which improve upon the use of

- Support effective prohibition on argument references persisting past the function return
- Class operations behaving like primitive type operations
- Documentation or enforcement of intent that the parameter is not intended to be polymorphic

5. Additional Possible solutions

We propose two additional solutions for consideration if something like Lawrence’s proposal does not work out:

1. User-defined pointer grouping
2. Newtype derived from fundamental types

5.1 User-defined pointer grouping

This proposal expands on restrict aliasing and allows the user to explicitly partition alias sets. It can use generalized attributes as a straw man for syntax.

For example, in restrict the current syntax:

int *restrict p = &i;

can be expanded as follows:

int * [[aliasGroup(Foo)]] p1 = <expr>
int *[[aliasGroup(Bar)]] p2 = <expr>
int * p3 = <expr>

*p2 would normally alias *p1, however, here because they don't share an alias group we would assume they don't alias. *p1 should alias *p3 because a pointer which specifies no alias group should be assumed to alias all alias groups and in many cases that's needed for correctness. In this way it's easy for programmers to take existing valid programs and tweak alias sets on only specific pointers to improve performance while preserving program correctness. The aliasing promise would be static for the life of the program, and not bound to any particular scope. This makes it easier for compiler vendors to implement. This proposal can also be implemented consistently in both C and C++.

We can also offer a shortcut for "one-off" alias groups to designate a pointer as disjoint from any other pointer which also specifies grouping. In a way, thinking of these as disjoint groups is more natural for users whereas thinking of them as alias group is more natural for implementers.

The attribute could be specified for any pointer type and also applied to member pointers. The attribute can also be embedded in a typedef. The attribute would not affect mangling and collide with non-attribute pointers of the same type.

We will also need to carefully specify how this aliasGroup attribute will be passed through to functors, lambdas, and templates.

This approach has the benefit that it:

- Enables explicit user control of aliasing
- Works for all cases
- Resolves many of the difficulties of restrict-like semantics with C++.
- Is lightweight and simple to adapt

5.2 Newtype derived from fundamental types

A general solution is based on the notion of generating a new type from existing types.
These new types allow one to declare a type which has the storage characteristics of one type, but type checks as an entirely different type. This proposal contains syntax and semantics for adding newtype to the C++ language.

Users employ a number of current solutions but they have their drawbacks.

- Wrapper types are common, but the members still alias the underlying types not to mention much boiler plating
- Restrict in C++ issues have been discussed in previous sections
- Primitive types are indivisible and comes with much attached properties such as implicit constructors, conversion operators
- Some compilers have implemented pragma disjoint to directly say that these pointers are restricted for the global scope of the program
- Enums can also act as inefficient wrappers in some cases and C++11 scoped enums do solve some of the issues but also do not go far enough.

There are several potential advantages to newtype constructs:

1. The base type and the new type may be aliased in an entirely different way, increasing optimization possibilities.
2. The two types are incompatible in the type system by default, allowing modular design without the cost of a wrapper type.
3. A user can use newtype to control the permitted actions of a primitive type, such as selecting type coercion.
4. The registerization characteristics of the underlying type are preserved
5. An efficient wrapper type allows for more efficient template metaprograming

To achieve this, we propose a new keyword (newtype) as a straw man placeholder to be added to the language, and a new form of declaration is created.

[template <typename T>] newtype NotInt : [public | private | protected] T [= default | explicit | deleted]
[{
  // only non-virtual member functions, no data or reference members (something like pod)
  // this->value has all of the operations of T, but only aliases with the new type, and implicitly converts to/from T prvalues.
  // inside members, this->int will give you access to the thing as an int (but not aliased as an int)
  // this->int aliases as NotInt, but acts in the type system as int (so we can modify it, etc.)
}] [optional variable name];

The settings for default, explicit, deleted can be used to decide on the actions on implicit/explicit constructors and conversion operators, as well as whether other form expression operators are allowed:

Newtype	Implicit ctor & conv	Explicit ctor & conv	Expression ops (+=, +)
Default	OK	NA	OK
Explicit		OK	OK
Deleted	User defined	User defined	User defined

Since this uses inheritance relationship, the keywords public, private and protected have their usual meaning.

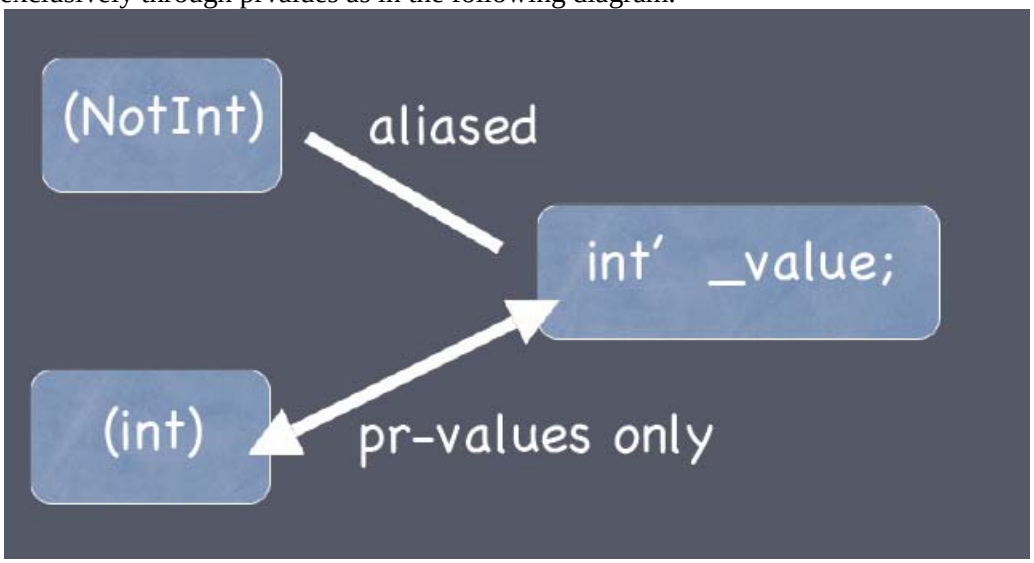
Some examples of constructions of newtypes are:

```
newtype NotInt : int;
NotInt k(5); // call ‘default’
             // int explicit ctor
int foo(int);
foo(k); // calls operator int()
```

```
NotInt bar(NotInt);
bar(k); // passes as an ABI int

int baz(int&);
baz(k); // Error! no conversion
```

Specifically, we wish to focus on the advantages for aliasing and the semantics of newtype. This design introduces a compiler-generated underlying type. The user-defined newtype only aliases itself and the underlying type. But the user-specified base type aliases neither the underlying type nor the user-defined newtype. The interaction between the base type, the underlying type, and the user-defined newtype occur exclusively through prvalues as in the following diagram:



It is possible to cast between the types through a new keyword alias_cast <New, Old> (…). The types can be split with newtype, and alias_cast acts as a bridge between them. They can be implemented as logical copies and are usually wrapped into the library.

Here are some examples of their usage:

```
newtype D1: double;
newtype D2: double;
void foo(D1 *x, D2 *y) {
  //D1/D2 can be assumed not aliased
  …
}

newtype SafeFloat : public double = explicit;

SafeFloat x(5.0); // explicit construction, ok
double d(x); // explicit conversion, ok

x + 5.0; // no impl conversion from 5.0 to SafeFloat
x + SafeFloat(3.0); // explicit construction, ok
x + SafeFloat(3); // explicit conversion, ok
```

x + d; // no impl conv
d = x; // no impl conv (from SafeFloat to double)

// public, so we can access the underlying value
d = x.value; // allowed (prvalue)
x.value = 1.0; // not allowed (prvalue)
// note that pointers to x cannot alias (double)
Newtype essentially hijacks similar syntax as strong enum and offers advantages which include:

- user-controlled type-based aliasing without sacrificing ABI optimizations for platform types
- better error messages
- easier to detect semantics errors without sacrificing performance
- customized behavior of platform types
- incremental adoption (orthogonal syntax based on strong enum)

N3538 also adds a solution for the "parameter ABI issue" which says that passing by value is now sometimes faster than passing by reference even if no copy is necessary.
The newtype mechanism can be used to implement modified behaviour with the same ABI impact as it's base type. The same holds for Walter's opaque typedefs.

It does not solve the parameter ABI issue in that it is still up to the user or library to know which is better: implement as pass-by-value or as pass-by-const-reference.
As far as I can tell, the "parameter ABI issue" can be solved by providing a library traits interface which provides that information.

This proposal is similar to Walter Brown's N3515 [12]. The concepts expressed by the opaque typedefs is vastly similar to the newtype proposal in terms of the customizability of operations.
There is a delta on the choice of reusing access specifiers (which we considered and rejected for the newtype proposal) and on the availability of "explicit" conversions.
The major difference is in the handling of the actual aliasing between the types under N3515 and under newtype.

The newtype proposal would work nicely under the N3515 syntax with the additon of "new" to the
access-specifier
portion of the grammar in that paper.

using NotInt = new int;

The underlying type for the purposes of N3515 with this addition is no longer "int", but the int' we discuss in the newtype proposal, which we can name NotInt::int.

5.2.1 Alising under newtype or opaque typedef

Some form of pointer type casting (eg, <alias_cast>) to the opaque type would be necessary for some uses like array stripes. The addition of alias_cast adds support for the RAII pattern which inserts the barriers allowing for direct control of aliasing.

We can use the RAII pattern to insert the code-motion barriers which conceptually are similar to N3538's copy of temporaries. We would like to evolve that idea into a smart-pointer style interface.

This closely matches, in effect, the proposal to solve the above problem in Section 3 by inserting the copy input and output in the constructor/destructor.

The proposed interface looks like this:

```
template <class To, class From>
std::vcopy_ptr<To, From> std::make_vcopy(From *q);

auto p = std::make_vcopy<typename new double>(q);
```

When we use this solution for the use-case in case would look like:

```
struct DoIt {
  DoIt(double *p, double *q, double *r) : p_(p), q_(q), r_(r) { }
  std::vcopy_ptr<typename new double> p_;
  std::vcopy_ptr<typename new double> q_;
  std::vcopy_ptr<typename new double> r_;

  // ...
};
```

The constructor and destructor for DoIt will call the corresponding std::vcopy_ptr special member (which implements the barriers).

Notice the "typename new double", this is an introduction of an anonymous new-type.
Whether two anonymous new-types are the same is determined in the same manner as other unnamed types (e.g., closure types).
i.e., is_same<typename new double, typename new double> == false

Caveat: This means that in the above:
```
  std::vcopy_ptr<typename new double> p_, q_, r_;
```
would not have the desired property.

I am toying with proposing that:
```
  std::vcopy_ptr<typename auto new double> p_, q_, r_;
```
would work as expected under the static type system.

We note that in a recursive call situation:
```
void foo(/* ... */) {
  using NewInt = new int;
  typename new double *p;
  NewInt *q;
  // ... contains recursive call
}
```

The expressions *p and *q in the caller should be considered to possibly alias the respective expression in the callee in the absence of additional information.
Using static type-based aliasing, this can be alleviated with the use of a template parameter representing the call depth.

Handling the array-striping case under this framework looks like this:
```
extern double *rawStrip;
struct CrossStrip { typename auto new double a, b, c; };
auto longStrip = std::make_vcopy<CrossStrip>(rawStrip);
// use longStrip ...
```

In essence, this proposal provides an unscoped, type-safe aliasing framework with the option of scoping introduced by the use of std::make_vcopy.
Replacing make_vcopy with reinterpret_cast is sufficient to avoid the code-motion barriers if it is known that it would be safe.

It is more heavy weight than the aliasGroup proposal but has greater flexibility. The aliasGroup feature seems lighter weight and adaptable immediately to current attribute syntax as it does not change the type system. However, changing the type system is an advantage not a because it would keep aliasing and types closely coupled which helps programmers understand the meaning. Types are aliased with themselves and their base types and some specific compatible types, but not with sibling types derived from the same base type or with incompatible types.

Conclusion

This paper offers feedback to N3538 and offers a point of proposed collaboration with the authors of N3538, and anyone else interested in collaborating on improving the performance of C++ through improved aliasing.

This paper analyzes the difficulties of adding restrict-like semantics to C++ and proposes two additional solutions for discussion. These solutions are the aliasGroup proposal for defining grouping of aliasing using an attribute, and the derivation of a newtype with defined semantics for augmenting the control of type-based aliasing.

Acknowledgement

This proposal is the culmination of work by at least the following and many others:
Kit Barton, Shimin Cui, Roland Froese.

Reference

[1] WG21 N3538 : Pass by Const Reference or Value, Lawrence Crowl : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3538.html

[2] WG14 N1409 : aliasing and effective type as it applies to unions/aggregates, Clark Nelson: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1409.htm

[3] WG14 N1520 : Fixing the rules for type-based aliasing, Clark Nelson : http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1520.htm

[4] C++-std-ext reflector discussion

[5] Microsoft: http://msdn.microsoft.com/en-us/library/8bcxafdh%28v=vs.80%29.aspx
[6] GCC: http://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html
[7] IBM: http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.cbclx01%2Frestrict_type_qualifier.htm
[8] Intel (EDG): http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/optaps/common/optaps_vec_use.htm
[9] HP(EDG): http://h30097.www3.hp.com/cplus/cxx_ref.htm?jumpid=reg_r1002_usen_c-001_title_r0010#desc
[10] Solaris: http://docs.oracle.com/cd/E18659_01/html/821-1383/bkana.html#indexterm-1007
[11] Pragma disjoint: http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.cbcpx01%2Foptpragm.htm
[12] WG21 N3515: Toward Opaque Typedefs for C++1Y, Walter Brown: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3515.pdf
[13] private communication with Clark Nelson
[14] private communication with Darryl Gove