Document Number: N3988
Date: 2014-05-23
Authors: Hal Finkel, hfinkel@anl.gov
Hubert Tong, hstong@ca.ibm.com
Chandler Carruth, chandlerc@google.com,
Clark Nelson, clark.nelson@intel.com,
Daveed Vandevoode, daveed@edg.com,
Michael Wong, michaelw@ca.ibm.com
Project: Programming Language C++, EWG
Reply to: michaelw@ca.ibm.com
Revision: 1

Towards restrict-like aliasing semantics for C++

Abstract

This paper is a follow-on to N3635 after a design discussion with many compiler implementers at Issaquah, and afterwards by telecon, and specifically proposes a new alias_set attribute as a way to partition type-based aliasing to enable more fine grain control, effectively giving C++ a more complete restrict-like aliasing syntax and semantics.

Changes from previous papers:

N3988: this paper updates with syntax and semantics
N3635: Outlined initial idea on AliasGroup which became alias_set. The alternative of newType was rejected as too heavy weight of a solution.

1. Problem Domain

There is no question that the restrict qualifier benefits compiler optimization in many ways, notably allowing improved code motion and elimination of loads and stores. Since the introduction of C99 restrict, it has been provided as a C++ extension in many compilers. 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. Given the existing compiler support, we need to provide a solution with well-defined C++ semantics, before use of some *common subset* of C99 restrict becomes widely used in combination with C++11 constructs. Some ill-defined interactions between restrict and C++ are:

- restrict class members and indirection with the “this” pointer; effect of restrict on references
- Overload resolution and template argument deduction in the presence of restrict qualifiers
- The effect of lambda captures of restrict-qualified entities

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.

2. Compiler implementations and related work

Some initial Internet research indicates that the restrict keyword (or its variants with underscores) is accepted by every major compiler:

- Microsoft [5]
- GCC [6]
- IBM [7]
- Intel (EDG) [8]
- HP (EDG) [9]
- TI [15]
- Clang [17]
- Nvidia (nvcc)

Additionally, IBM's #pragma disjoint [11] asserts non-aliasing with specific pointers. Solaris [10], does not provide the keyword, but does provide a command line option that modifies all pointers in a file. This indicates there is wide-spread demand for restrict-semantics, or at least an appetite for the performance benefit offered by a facility similar to restrict. Without standardizing and improving the existing C99 restrict facility in C++, 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.

Related Proposals:

Aliasing long been a problematic area in both C and C++, and C++ essentially adapts its aliasing rules from C in Clause 6.5p7. Several papers by Clark Nelson have attempted to fix the aliasing rules [2][3]. Work on improving type-based aliasing should continue, however, the discussion in these papers on effective types is orthogonal to our proposal.

The solution proposed in this paper does not conflict with the proposal in N3538, and was partially inspired by it, but provides a superset of N3538's expressive capabilities while limiting language changes to the addition of a new attribute.

3. Issues with restrict in C++ and C

C99 restrict qualifier support is not part of C++11. At the Mont Tremblant meeting [?], when reviewing C99 feature inclusion in C++, restrict was considered. It was noted in the record that the committee was awaiting a proposal, but none came forward. However, it was one of the few features that was considered as favorable for C++.

That having been said, C99 restrict has some undesirable aspects. Furthermore, it currently has ambiguities, even in C, because it is unclear exactly how it applies to struct members [13].

Another issue is how to use restrict when it is desired to consider only one pointer in a chain of pointers to have restrict-like semantics. The semantics of restrict in C is such that non-restrict pointers to restrict pointers may alias each other, and combined with how restrict in C is defined in terms of access based on

a pointer object, accessing a restrict-qualified pointer indirectly via non-restrict pointers provides little information to the compiler.

C99 restrict syntax in C++ has even more issues to resolve. restrict is a C99 feature and was never designed to work in class abstractions. For example, it is unclear how the qualifier will be carried through functors and templates. It is specifically unclear 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);
}
```

It is also unclear if it participates in the 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
```

Moreover, as highlighted by the example below, the guarantees provided by restrict are often inappropriate, or too weak, when used on member variables. 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.

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, specifically, 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);
}
```

In addition, the above loop, when properly aliased, can be vectorized automatically without explicit directives.

We wish also to solve the issue 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 noted by our colleague [?]:

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.

4. Goals of this design

Given the above discussion on issues, it is the intention of the solution to resolve or cover the following cases:

- Variables, functions, static data members
- Members variables of struct/unions/class, for every level of nesting
- Overlapping array members and strides including multi-dimensional arrays
- Support lambdas, functors, templates
- Indirect references through implicit this pointer
- Prevent escaping of restrict pointer values inside functions
- Support every level of nested pointer operator

5. Design solution

At the Sept. 2013 Chicago meeting, feedback from the initial presentation of N3635 indicated broad support for allowing the user, using generalized attribute syntax, to explicitly partition alias sets into disjoint groups even within type-based aliasing rules. There was guidance that it should not carry through typedefs and that it should not affect the type system. Accordingly, we do not propose the use of type attributes, but instead propose using attributes in such a way that they do not affect the type. In addition, as the implementation of this feature requires compiler backend support, we needed to get buy-in from major compiler backends.

In Feb 2014 Issaquah meeting, a group [16] met to work on a solution, again based on the AliasGroup design in N3635, but now expanded and renamed alias_set. This solution creates an aliasing attribute syntax to describe the partitioned sets for even complex nesting and member components. This proposal is discussed here in a quasi-grammar and initial draft wording, to give an indication of the direction. We look for feedback from other vendors beyond the current authors list, including GNU, Microsoft, Embarcadero, and Solaris.

Proposal with pseudo draft Standard wording:

Add the following subsection of 7.6 **[dcl.attr.grammar]**:

7.6.x Alias-Set attribute [dcl.attr.as]

The attribute-token alias_set restricts the set of objects that a pointer or reference variable may be used to access. It may occur multiple times in each attribute-list and the attribute-argument-clause shall have the format:

```
as-attribute-argument-clause:
  ( as-set-list )

as-set-list:
  as-set-list ; as-set-specifier
  as-set-specifier

as-set-specifier:
  as-set-name as-predicate_opt
  *
  ...

as-predicate:
  [ assignment-expression ]
```

as-set-name:
  qualified-id_opt

The attribute may be applied to any declarator-id in a declarator. When applied to the declarator-id in a function declaration, the attribute set applies to the function's return value, and the return type must be a pointer or reference type. Otherwise, the attribute set applies to the associated parameter or variable. Let V be the return value, parameter or variable. V must either be a reference or a multi-level mixed pointer (4.4). If V is a reference, then the alias set list must have at most one specifier, which may not be the ... specifier. The alias set specifier restricts the universe of objects to which the reference may bind. Otherwise, the alias set list may have at most n specifiers (for the n that defines the multi-level mixed pointer type). The first specifier restricts the use of V by limiting dereferences through it to objects in a restricted universe, the second restricts the use of *V similarly, the third restricts the use of **V, and so on. If the ... specifier is present, it must be the last specifier in the specifier list and the specifier list must contain at least one other specifier. When the ... specifier is present, then the list is treated as if it had n specifiers where the last specifier in the list prior to the ... is used in place of the ... and for any others that are missing. When the * specifier is present, the universe identified by the specifier is unspecified.

Each aforementioned restricted universe of objects is identified by an alias set specifier consisting of an optional name, which may be any qualified-id resolvable as specified below, and an optional predicate. If neither is provided, then the universe is unspecified, but the disjointness restrictions specified in 3.7.4.1 [basic.stc.dynamic.allocation] paragraph 2 for the return values of library allocation functions apply [Note: see core issue 1338]. If only the name is provided, the fully-qualified name identifies the universe. The name shall resolve to a variable, enumerator, type or namespace, in that order of preference. If the name is not provided, then the universe is identified by the tuple of the type of the predicate expression and the value of the expression. If both the name and predicate are provided, then the tuple of the fully-qualified name, the kind of the name (variable, type, etc.), type of the predicate expression and value of the expression identify the universe. Each identified universe is disjoint from all other universes. When comparing two universes where the relevant identifying tuples contain dynamic predicates, the expression P1 == P2 shall be used in the relevant evaluation context (specified in the next paragraph).

When the attribute is applied to the declarator-id in a function declaration or applied to the declarator-id of a parameter-declaration in a function declaration or lambda, for the purpose of name resolution, the name of the universe and names contained within the predicate expression are resolved as though the predicate was the expression in an expression-statement, and this statement occurred first within the function or lambda body. When applied to the declaration of a global member variable, name resolution occurs within the context of an unnamed, parameterless global or member function in the same scope as the declaration. When a predicate expression is provided, the value used to partially identify the universe is the value of the predicate expression sequenced immediately prior to the evaluation accessing the restricted value. Whether the predicate expression is evaluated prior to any particular access of the restricted value is unspecified.

When multiple alias set specifiers apply to a particular pointer object or reference, the restriction is to the union of all provided universes.

Application of Restrictions:

Exactly how to specify when and how the restrictions apply is currently undecided. There are a few possible models:

Model One:

At the points at which a predicate expression might be evaluated, no other pointer values or references in that scope not specifically restricted to a specified universe, but that are unconditionally dereferenced assuming that this evaluation point is reached, may have the address of or bind to an object in the specified universe [Note: Maybe unconditionally is too strong?].

[Note: define “derived from” in terms of a decidable set of syntactic elements, similar to that used in 3.2 paragraph 3 for ODR uses for pointer and reference types.]

When an object of pointer or reference type is derived from one or more pointer or reference values restricted to some set of universes, the value of that pointer object, or the object to which the reference is bound, is restricted to the union of those universes. The restrictions on the accesses via a pointer's value apply in relation to accesses which are indeterminately sequenced with the dereference (use by the unary *, -> or *-> operators). For reference variables, the restrictions only apply after any applicable initializer is evaluated and the reference is bound to the resulting value. The restrictions of a variable to a particular universe only applies in the scopes where the alias set name is visible and accessible (in the public/private sense). For unnamed universes, the restrictions apply only to the scope of the variable to which the attribute is applied.

Model Two:

If the value of a pointer or the address of a reference is used to form a glvalue expression which is used to access the object which the glvalue denotes, and the use of the pointer value or reference is restricted as described in this section, then the following applies:

- the access to the object is understood to potentially access any of the universes associated with the pointer values or references used to form the glvalue expression
- if access to the object occurs via a glvalue which is not understood to potentially access at least one of the universes considered in the previous point, then the behavior is undefined unless if the two accesses are separated by an appropriate transfer event

In the dynamic execution of the program, a side-effect of the assignment of an address value which is understood to potentially access a set of universes to a pointer whose use is restricted for another set of universes (or if, explicitly specified, to the generic universe) and the equivalent case of reference binding, is a transfer event which acts as suitable separation between an access of the same object via the former set of universes and the latter set of universes.

Motivating examples

The examples below illustrate how the syntax works. It also provides commentary on the desired conclusions that a compiler may safely make. The precise semantics applied to reach the conclusion is left unspecified.

[Example:
```
 [[ alias_set() ]] void *my_malloc(size_t); // The return value is the address of an object that is disjoint from all others (just like malloc).
 [[ alias_set(tagX) ]] T *x;
```

```
 x = my_malloc(n);
 v = *x; // x is restricted to the universe of tagX (and the compiler may infer that it is disjoint from any other pointer in tagX as well).

 [[ alias_set(tagA), alias_set(tagB) ]] int *x; // objects accessed using x are restricted to the union of the 'tagA' universe and 'tagB' universe.

 struct tag;
 void foo([[ alias_set(a) ]] double *a, [[ alias_set(a) ]] double *b, [[ alias_set(tag) ]] double *c);
 // The pointer values of a and b, and any derived from a or b might refer to the same object. The pointer value of c can only refer to objects in the 'tag' universe (assuming a dereference). However, if foo is inlined into its caller, the alias_set guarantees conferred by the alias_set a still only apply to aliasing comparisons between dereferences that came from the inlined function, not between those and dereferences in the caller.

 class container {
 protected:
   struct as {};
   [[ alias_set(as[this]) ]] int *start, *end;
   // These pointers, or those based on them, might refer to the same objects. No other pointers dereferenced by functions that could access container::as will do so.
 };

 [[ alias_set(tagY) ]] T* bar([[ alias_set(tagX) ]] T * x) {
    return x; // this will cause undefined behavior if the return value is ever dereferenced in a context where it is restricted to the tagY universe.
  }

  // assume that std::vector's internal pointers have an alias_set as in the preceding container example using some tag that is private to std::vector.
  void foo([[ alias_set(tagX) ]] T *x, std::vector<T> &V);

  [[ alias_set(tagX) ]] *x = &V[0];
  foo(x, V); // this is valid; the alias_set guarantees from std::vector's only apply to aliasing comparisons where the vector's alias_set tag is in scope and accessible. So if two of std::vector's operator [int] were inlined, then the two dereferences contained therein could be dealiased, but nothing could be said about the dereference within the std::vector member function and the *x from within foo.
]
```

Summary of Benefits:

This approach has the benefit that it:

- Enables explicit user control of aliasing
- Replace based-on, which is often undecidable in practice, with an ODR-use-inspired mechanism that is easily solvable
- Resolves many of the difficulties of restrict-like semantics with C++
- Is lightweight and simple to adapt
- Can be ignored but does not change the correctness of the program

Conclusion

This paper analyzes the difficulties of adding restrict-like semantics to C++ and proposes an alternative attribute-based mechanism to express pointer/reference aliasing properties and allow the compiler to perform the optimizations that users demand. This mechanism does not require any core language syntax changes and/or changes to the type system, is more expressive than C99's restrict, and naturally integrates with modern C++ features.

Acknowledgement

This proposal is the culmination of work by at least the following and many others:
Chris Bowler, Yaoqing Gao, Ian Mcintosh, Chris Lord, Sean Perry, Kit Barton, Shimin Cui, Roland Froese. Tom Plum, Torvald Riegel, Olivier Geroux, Jeff Snyder, hcedwar@sandia.gov,

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
[15] https://processors.wiki.ti.com/images/f/ff/Bartley%3DWiki_1.1%3DPerformance_Tuning_with_the_RESTRICT_Keyword.pdf
[16] Participants: Tom Plum (Plumhall), Chandler Carruth (Google), Carter Edwards (SNL), Olivier Geroux (Nvidia), Torvald Riegel(Redhat), Clark Nelson (Intel), Daveed Vandevoode(EDG), Hal Finkel(ANL), Jeff Snyder (Morgan Stanley), Michael Wong (IBM)
[17] clang Users manual: http://clang.llvm.org/docs/UsersManual.html