[[assert: std::disjoint(A,nA, B,nB)]]: Contract assertions as an alternate spelling of ‘restrict’

Document Number: P1296R0
Date: 2018-10-08
Reply-to: Phil Miller <phil.cpp@intensecomputing.com>
Authors: Phil Miller, Justin Szaday
Audience: Compiler implementers, for feedback on whether this syntax is suitable input for obviating alias analysis, LWG or wording review and adoption

Introduction

There have been many calls over the years for code to have a way to signal compilers that certain objects do not alias, allowing the compiler to optimize more aggressively. These include at least

- C99's `restrict` keyword and associated `__restrict__` extensions in C++ compilers
- the alias sets proposed in N4150
- `may_alias` attributes
- IBM XL's `#pragma disjoint`
- `restrict_ptr` proposals

However, specification of how C’s `restrict` extends to C++-specific constructs has seemingly been comparatively fraught with difficulties, and hence has not proceeded.

With the addition of standard contract syntax and semantics to the C++ language, I believe that similar effects can be achieved with no language impact and greater clarity for programmers by adding a predicate to the standard library, tentatively named `disjoint` here. This predicate could be used in `[[expects]]` and `[[assert]]` contracts to convey the desired information.

Related Proposals

P0856 (Hollman, Edwards, Trott. “Restrict Access Property for mdspan and span”) provides complementary elaboration of the motivations for this effort, and why it should be pursued as a library rather than language feature.

Present Proposal

Add a free function defined as follows to namespace std, in an appropriate header TBD:

```
template <typename T, typename U>
bool disjoint(const T* pt, size_t nt, const U* pu, size_t nu)
{
  intptr_t bt = pt, et = pt+nt;
  intptr_t bu = pu, eu = pu+nu;

  return (et <= bu) || (eu <= bt);
}
```

Requirements:

- The pointers pt and pu are valid.
- The expressions pt+nt and pu+nu are valid. (i.e. they point to elements within the same array as pt and pu, respectively, or to one past the end, including the case where pt is a pointer is to a non-array object and nt is 1 (or pu and nu, respectively); also, they do not result in arithmetic overflow)

Motivating Data

To illustrate the substantial value of standardizing a way for developers to convey the absence of alias-created hazards to a compiler transforming a loop, we need to evaluate code in which relevant arguments do not alias, comparing the cases where the compiler is or isn't provided with that knowledge.

These data were provided by Justin Szaday of the University of Illinois, using a system he helped develop, [LORE: LOop Repository for the Evaluation of compilers](). I asked him to search for all entries in the repository that used `__restrict__` as ingested. At my request, he developed and applied a transformation to remove the `__restrict__` qualifiers, and ran the stored code benchmarks for each such loop on the original and modified versions of the code.

There are 2507 codelets in the repository that use __restrict__. Among the benchmark suites from which they were extracted, their prevalence in each suite is as follows:

```
{'spec2006':    0.66, 'scimark2':        0.89, 'netlib': 0.88,
 'spec2000':    0.72, 'TSVC':            0.12, 'Kernels':  0.26,
 'ASC-llnl':    0.12, 'machinelearning': 0.87, 'NPB': 0.13,
 'cortexsuite': 0.85, 'FreeBench':       0.71, 'GitApps': 0.91,
 'Fhourstones': 0.67, 'libraries':       0.73, 'mediabench': 0.84,
 'ALPBench':    0.97, 'polybench':       0.03}
```

This shows that among source code that expects to be performance sensitive, there is strong demand for and willingness to use a means to explicitly inform compilers about the absence of aliasing as an impediment to optimization.

Of 1861 loops evaluated with ICC 17.0.1 with -O3 optimization, 45% of the loops were 1.15x or more faster with restrict than without it, and the geometric mean speedup of having it was 2.59x. 50% of the loops were minimally affected by removing restrict (within a 0.15x difference) and, curiously, 5% of the loops were sped up by removing restrict. The most extreme values are speedups of 28x, 33.25x, 35.5x, 42.4x, 48.4x, and 57.2x, and slowdowns of 0.27x and several of 0.32x.

Of the 1939 loops evaluated with GCC version 6.2.0 with -O3, 734 (38%) experienced statistically significant speedup by having restrict and 155 (8%) significant slowdown (Welch's t-test given unequal variance, 2 tailed, $p < 0.05$). The geometric mean of the execution time ratio among loops with speedup was 2.385, and among loops with slowdown was 0.766. The most extreme speedups were 41x, 39x, 39x, 38x, and 33x, and slowdowns of .26x, .36x, and .40x.

Evaluation of Clang 6.0.1 showed minimal results. We believe that this is the result of Clang generating runtime checks for aliasing and code paths optimized for each alternative, but have not confirmed this by inspection of the generated object code at this time.

	Loops	Sped up	Mean Speedup	Slowed	Mean Slowdown
GCC	1939	734 (38%)	2.39x	155 (8%)	0.766
ICC	1861	843 (45%)	2.59x	94 (5%)	0.61

From the above, we conclude that providing this information is very valuable to compilers, and thus to developers and users of the given and resulting code.

Further analysis and likely a conference or journal paper will explore what transformations are enabling the various degrees of speedup. For instance, smaller speedups are likely just vectorization or similarly fine-grain optimizations, while larger speedups are likely the result of major transformations like loop interchange that enable drastically better cache behavior.

Design Considerations

The basic design of this predicate will refer to ranges of memory locations, without regard for the types involved, to avoid hang-ups seen with adopting `__restrict__` itself in C++ about issues like class members, views as vectors or scalars, etc.

I consider it a feature that this approach explicitly confines its statement about each input to a particular range of locations (a single object, or some number of whole objects), rather than

“based on” pointer expressions from the C standard. As a quality of implementation matter, compilers can use this information to explicitly inform developers when the provided disjointness declarations are insufficient to enable transformations blocked by potential aliasing.

I choose to offer an interface using (pointer, count) pairs rather than (begin, end) pairs because the expected use centers on arrays of counted objects, or where the count is more concise to obtain than the end address.

E.g.

```
char *strcpy(char *dest, const char *src)
[[expects: disjoint(src, strlen(src), dest, strlen(src))]]
```

(As an aside, this illustrates that init-statements would be useful to have available in contract attribute contexts)

```
template<typename T>
void template_vector_axpy(double a, std::vector<T> &x, const
std::vector<T> &y)
[[expects: disjoint(x.data(), x.size(), y.data(), y.size())]]

// Multiply m*n matrix by n*1 vector
void matrix_vector_multiply(int m, int n, const double *matrix, const
double *vector, double *output)
[[expects: disjoint(matrix, m*n, output, n) && disjoint(vector, n,
output, n)]]
// Note that const doesn’t help, because it only enforces that we
don’t write through *matrix or *vector, not that the pointed-to
values don’t change otherwise
```

Overloads

We really only *need* the very basic general case, but in the long run, I expect having various other overloads also specified is desirable. I think it would be desirable to allow user-defined overloads that involve at least one user-defined type, but it would have to be a template or inline function from which compilers could similarly derive the substantive non-aliasing conclusions from visible address comparisons, since I do not intend to prescribe specific semantics for *all* functions named ‘disjoint’ used in contract expressions. My intention is to suggest a shared vocabulary convention.

```
// Basic, relatively general case
template <typename T, typename U>
bool disjoint(const T* pt, size_t nt, const U* pu, size_t nu)
{
```

```
  intptr_t bt = pt, et = pt+nt;
  intptr_t bu = pu, eu = pu+nu;

  return (et <= bu) || (eu <= bt);
}

// E.g. `t` does not refer to any recursive member of `u`, nor vice
versa
template <typename T, typename U>
bool disjoint(const T& t, const U& u)
{
  intptr_t at = std::addressof(t);
  intptr_t au = std::addressof(u);

  return disjoint(at, 1, au, 1);
}

// Useful in things like sum(double *A, int n, double &s) for
// distinguishing output variables from the input
template <typename T, typename U>
bool disjoint(const T* pt, size_t nt, const U& u)
{
  intptr_t au = std::addressof(u);

  return disjoint(pt, nt, au, 1);
}

For std::span defined in P0122, similarly:

template <typename S, typename T, typename U>
boot disjoint(std::span<const T> s, const U& u)
{
  intptr_t au = std::addressof(u);

  return disjoint(s.data(), s.size(), au, 1);
}
```

Naming

WIth the above in mind, I would be unsurprised by bikeshedding of the name to something more descriptive like `disjoint_memory`. I believe that whatever name is chosen should have the same non-aliasing positive/negative sense as ‘disjoint’, to avoid creating a ubiquitous need for negation in every expression where it’s used as intended.

C Compatibility

Whatever we do here, it would be desirable to maintain the possibility of importing compatible functionality into the C standard as well. It would ideally pair with adoption of a proposal to add contracts or other compiler-meaningful assertions in some form to the C standard.

The obvious adaptation is a non-template prototype along the lines of `_Bool disjoint(const char* pt, size_t nt, const char* pu, size_t nu)` This would work reasonably well, with an expression like `sizeof(element_type)*n` generally appearing as the count arguments. C's overload mechanism could be used to specify versions for each basic type, but as I understand it would not work for arbitrary user-defined types.

Usage applicability survey

Sampling a few dozen distinct uses of `__restrict__` on Github, I found no cases that this predicate and an `[[expects]]` or `[[assert]]` contract would not cover the intended meaning with only a reasonable amount of verbiage.

Furthermore, in many cases, the existing uses of `__restrict__` actually over-specify the desired meaning. For instance, a function with many input arrays and a single output array may designate them all `__restrict__`, when the relevant characteristic is that each of the inputs is disjoint from the output.

Acknowledgments

David Hollman provided encouragement, review, and references to additional relevant work. Arthur O'Dwyer pointed out the possibility of arithmetic overflow in the expressions calculating the end pointers.