std::spaceship

(which doesn’t exist)

Motivating use case (LLVM)

```
/// array_pod_sort - This sorts an array with the specified start and end extent.
/// This is just like std::sort, except that it calls qsort instead of    using an inlined template.
/// qsort is slightly slower than std::sort, but most sorts are not performance critical in LLVM
/// and std::sort has to be template instantiated for each type, leading to significant measured
/// code bloat.  This function should generally be used instead of std::sort where possible.
///
/// This function assumes that you have simple POD-like types that can be compared with operator<
/// and can be moved with memcpy.  If this isn't true, you should use std::sort.
///

template <class IteratorTy>
inline void array_pod_sort(
    IteratorTy Start,
    IteratorTy End,
    int (*Compare)(
        const typename std::iterator_traits<IteratorTy>::value_type *,
        const typename std::iterator_traits<IteratorTy>::value_type *)) {
  // Don't dereference start iterator of empty sequence.
  if (Start == End) return;
  qsort(&*Start, End - Start, sizeof(*Start),
        reinterpret_cast<int (*)(const void *, const void *)>(Compare));
}
```

with some of the cruft removed

```
/// array_pod_sort - This sorts an array with the specified start and end extent.
/// This is just like std::sort, except that it calls qsort instead of    using an inlined template.
/// qsort is slightly slower than std::sort, but most sorts are not performance critical in LLVM
/// and std::sort has to be template instantiated for each type, leading to significant measured
/// code bloat.  This function should generally be used instead of std::sort where possible.
///
/// This function assumes that you have simple POD-like types that can be compared with operator<
/// and can be moved with memcpy.  If this isn't true, you should use std::sort.
///

template <typename T>
void array_pod_sort(T *start, T *end, int (*compare)(const T *, const T *))
{
  // Don't dereference start iterator of empty sequence.
  if (start == end) return;
  std::qsort(/* base   */ start,
             /* nelem  */ end - start,
             /* width  */ sizeof *start,
             /* compar */ reinterpret_cast<int (*)(const void *, const void *)>(compare));
}
```

Some example comparators

```
static int SrcCmp(const std::pair<const CFGBlock *, const Stmt *> *p1,
                  const std::pair<const CFGBlock *, const Stmt *> *p2) {
  if (p1->second->getLocStart() < p2->second->getLocStart())
    return -1;
  if (p2->second->getLocStart() < p1->second->getLocStart())
    return 1;
  return 0;
}

static int compareEntry(const Table::MapEntryTy *const *LHS,
                        const Table::MapEntryTy *const *RHS) {
  return (*LHS)->getKey().compare((*RHS)->getKey());
}

static int CompareCXXCtorInitializers(CXXCtorInitializer *const *X,
                                      CXXCtorInitializer *const *Y) {
  return (*X)->getSourceOrder() - (*Y)->getSourceOrder();
}
```

https://www.mail-archive.com/cfe-commits@cs.uiuc.edu/msg92289.html
Clang's r203293 "[C++11] Revert uses of lambdas with array_pod_sort."

This is a pretty common idiom

Particularly in C.

`strcmp`

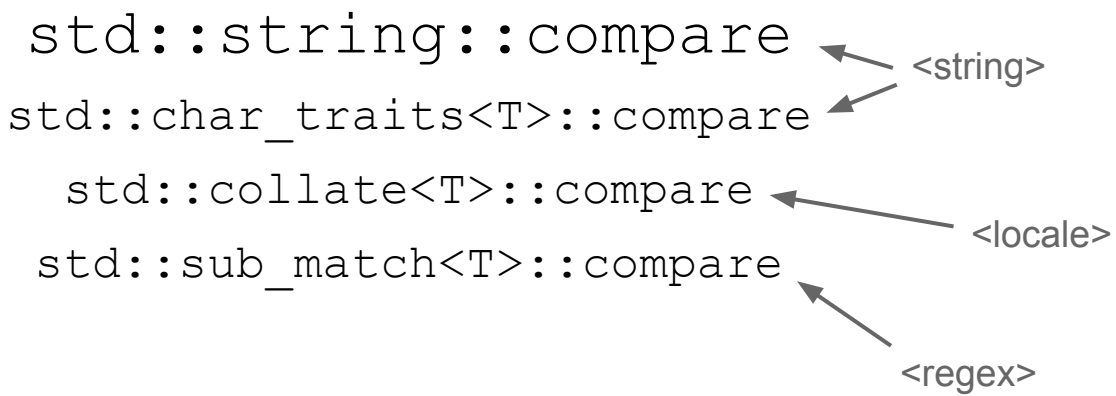
(`strcoll`, `strcasecmp`...)

`qsort`

`bsearch`

This is a pretty common idiom

But also (occasionally) in C++!



But look at the variety of these comparators!

```
if (a < b) return -1;                                    // the Java programmer's approach
if (a > b) return 1;
return 0;

return a.compare(b);           // otherwise known as "delegating the task to someone else"

return a - b;                                       // short and sweet, but can lead to bugs

return (a < b) ? -1 : (a > b);                                        // my personal favorite

return (a < b) ? -1 : (b < a) ? 1 : 0;     // the minimalist approach: uses only operator<

if (a != b) return (a < b) ? -1 : 1;                              // the extensible approach
return 0;
```

Wouldn’t it be nice if there were a
simple, unified way
to write comparators like these?

This is a solved problem

In Perl, Ruby, Groovy...

This is a solved problem

In Perl, Ruby, Groovy...

The spaceship operator

<=>

This is a solved problem

In Perl, Ruby, Groovy...

The spaceship operator

```
a <=> b
```

means

```
(a < b) ? -1 :
(a > b) ? +1 : 0
```

First attempt: LLVM to the rescue!

```
/// array_pod_sort_comparator - This is helper function for array_pod_sort,
/// which just uses operator< on T.
template<typename T>
inline int array_pod_sort_comparator (const void *P1, const void *P2) {
  if (*reinterpret_cast<const T*>(P1) < *reinterpret_cast<const T*>(P2))
    return -1;
  if (*reinterpret_cast<const T*>(P2) < *reinterpret_cast<const T*>(P1))
    return 1;
  return 0;
}
```

First attempt: LLVM to the rescue!

```
/// array_pod_sort_comparator - This is helper function for array_pod_sort,
/// which just uses operator< on T.
template<typename T>
inline int array_pod_sort_comparator (const void *P1, const void *P2) {
  if (*reinterpret_cast<const T*>(P1) < *reinterpret_cast<const T*>(P2))
    return -1;
  if (*reinterpret_cast<const T*>(P2) < *reinterpret_cast<const T*>(P1))
    return 1;
  return 0;
}
```

The problem is inefficiency.

First attempt: LLVM to the rescue!

```
/// array_pod_sort_comparator - This is helper function for array_pod_sort,
/// which just uses operator< on T.
template<typename T>
inline int array_pod_sort_comparator (const void *P1, const void *P2) {
  if (*reinterpret_cast<const T*>(P1) < *reinterpret_cast<const T*>(P2))
    return -1;
  if (*reinterpret_cast<const T*>(P2) < *reinterpret_cast<const T*>(P1))
    return 1;
  return 0;
}
```

The problem is inefficiency.

Two calls to `operator<` per comparison.

What if `T` is `std::tuple`?

libc++’s tuple comparison

```
template <size_t _Ip>
struct __tuple_less
{
    template <class _Tp, class _Up>
    bool operator()(const _Tp& __x, const _Up& __y)
    {
        return __tuple_less<_Ip-1>()(__x, __y) ||
            (!__tuple_less<_Ip-1>()(__y, __x) && get<_Ip-1>(__x) < get<_Ip-1>(__y));
    }
};

template <>
struct __tuple_less<0>
{
    template <class _Tp, class _Up>
    bool operator()(const _Tp&, const _Up&)
    {
        return false;
    }
};

template <class ..._Tp, class ..._Up>
bool operator<(const tuple<_Tp...>& __x, const tuple<_Up...>& __y)
{
    return __tuple_less<sizeof...(_Tp)>()(__x, __y);
}
```

2 tuple comparisons

=

$2n$ element comparisons

So what?

Who uses tuples for anything?

Who *compares* tuples?

Unfortunately, lots of people

http://vexorian.blogspot.com/2013/07/more-about-c11-tuples-tie-and-maketuple.html

http://stackoverflow.com/questions/10806036/using-make-tuple-for-comparison

http://stackoverflow.com/questions/6218812/implementing-comparision-operators-via-tuple-and-tie-a-good-idea

http://siliconkiwi.blogspot.com/2012/04/stdtie-and-strict-weak-ordering.html

http://oraclechang.files.wordpress.com/2013/05/c11-a-cheat-sheete28094alex-sinyakov.pdf

http://latedev.wordpress.com/2013/08/12/less-than-obvious/

http://wordaligned.org/articles/more-adventures-in-c++

And on the topic of adding a “spaceship function” to C++:

Generic compare function (Adam Badura)

Why aren't “tri-valent” comparison functions used in the standard library? (K. Frank)

The idiom we want to use in C++14

```
class MyClass {
    int a, b, c, d;
public:
    auto tied() const {
        return std::tie(a,b,c,d);
    }
    bool operator< (const MyClass& rhs) const {
        return tied() < rhs.tied();
    }
};


... array_pod_sort_comparator<MyClass> ...
```

The idiom we want to use in C++14

```
/// array_pod_sort_comparator - This is helper function for array_pod_sort,
/// which just uses operator< on T.
template<typename T>
int array_pod_sort_comparator (const T& a, const T& b) {
  if (a < b)
    return -1;
  if (b < a)
    return 1;
  return 0;
}
```

```
... array_pod_sort_comparator<MyClass>  ...
```

This is disastrously inefficient when `(a == b)`!

Twice as many comparisons as necessary!

How to fix it

We need a trivalent comparison function for tuples.

```
namespace std {

    template <typename T, typename U>
    int spaceship(const T&, const U&);

    template <typename... T, typename... U>
    int spaceship(const tuple<T...>&, const tuple<U...>&);

}
```

How to fix it

```
// The easy part.
//

namespace std {

    template <typename T, typename U>
    int spaceship(const T& x, const U& y)
    {
        return (x < y) ? -1 : (y < x) ? 1 : 0;
    }

}
```

How to fix it

```
// The easy part.
//

namespace std {

    template <class _Tp, class _Up>
    constexpr int spaceship(const _Tp& __x, const _Up& __y)
    {
        return (__x < __y) ? -1 : (__y < __x) ? 1 : 0;
    }

}
```

How to fix it

```
// The barely harder part.
//

namespace std {

    int spaceship(const string& x, const string& y)
    {
        int r = x.compare(y);
        return (r < 0) ? -1 : (r > 0);
    }

}
```

How to fix it

```
// The barely harder part.
//

namespace std {

    template<class _Cp, class _Tp, class _Ap,
                        class _Up, class _Bp>
    int spaceship(const basic_string<_Cp,_Tp,_Ap>& __x,
                  const basic_string<_Cp,_Up,_Bp>& __y)
    {
        int __r = __x.compare(0, __x.size(), __y.data(), __y.size());
        return (__r < 0) ? -1 : (__r > 0);
    }

}
```

// The hard part.

```
template <size_t _Ip>
struct __tuple_spaceship
{
    template <class _Tp, class _Up>
    constexpr int operator()(const _Tp& __x, const _Up& __y) const
    {
        int __r = __tuple_spaceship<_Ip-1>()(__x, __y);
        return (__r != 0) ? __r : spaceship(get<_Ip-1>(__x), get<_Ip-1>(__y));
    }
};

template <>
struct __tuple_spaceship<0>
{
    template <class _Tp, class _Up>
    constexpr int operator()(const _Tp&, const _Up&) const
    {
        return 0;
    }
};

template <class ..._Tp, class ..._Up>
constexpr int spaceship(const tuple<_Tp...>& __x, const tuple<_Up...>& __y)
{
    static_assert(sizeof...(_Tp) == sizeof...(_Up));
    return __tuple_spaceship<sizeof...(_Tp)>()(__x, __y);
}
```

The idiom we *should* use in C++14

```
class MyClass {
    int a, b, c, d;
public:
    auto tied() const {
        return std::tie(a,b,c,d);
    }
    bool operator< (const MyClass& rhs) const {
        return spaceship(*this, rhs) < 0;
    }
};

int spaceship(const MyClass& a, const MyClass& b) {
    return std::spaceship(a.tied(), b.tied());
}


... array_pod_sort_comparator<MyClass> ...
```

The idiom we *should* use in C++14

```
/// array_pod_sort_comparator - This is helper function for array_pod_sort,
/// which just uses "spaceship" on T.
///
template<typename T>
int array_pod_sort_comparator (const T& a, const T& b)
{
  using std::spaceship;

  return spaceship(a, b);
}
```

```
... array_pod_sort_comparator<MyClass> ...
```

This is efficient even when `(a == b)`!

Only as many comparisons as necessary!

Unfortunately...

std::spaceship is not part of C++14.

You know anyone on the standards committee?

```
end()
```