C++11 in the Wild

Techniques from a real codebase

What we'll cover

- The **Auto()** macro (slides 3–34)

  run arbitrary cleanup code at the end of a scope

- **make_iterable()** and **iterator_range** (slides 35–49)

  turn a pair of iterators into a container

Time permitting: • **std::spaceship** (slides 50–81)

compare any two objects strcmp-wise

Chapter 1.
The Auto() macro

A better OnScopeExit()

What we start with

```
void Mutate(State *state)
{
  state->DisableLogging();
  state->AttemptOperation();
  state->AttemptDifferentOperation();
  state->EnableLogging();
  return;
}
```

Oops, forgot all the error handling

```
bool Mutate(State *state)
{
  state->DisableLogging();
  if (!state->AttemptOperation()) return false;
  if (!state->AttemptDifferentOperation()) return false;
  state->EnableLogging();
  return true;
}
```

(Or, use exceptions for control flow if you want. You'll have the same problem.)

What we want to write

```
#include "auto.h"

bool Mutate(State *state)
{
  state->DisableLogging();
  Auto(state->EnableLogging());

  if (!state->AttemptOperation()) return false;
  if (!state->AttemptDifferentOperation()) return false;
  return true;
}
```

#include "auto.h"

Credits: Marko Tintor, Alex Skidanov, Arthur O'Dwyer

```
#pragma once

template <class Lambda> class AtScopeExit {
  Lambda& m_lambda;
public:
  AtScopeExit(Lambda& action) : m_lambda(action) {}
  ~AtScopeExit() { m_lambda(); }
};

#define TOKEN_PASTEx(x, y) x ## y
#define TOKEN_PASTE(x, y) TOKEN_PASTEx(x, y)

#define Auto_INTERNAL1(lname, aname, ...) \
    auto lname = [&]() { __VA_ARGS__; }; \
    AtScopeExit<decltype(lname)> aname(lname);

#define Auto_INTERNAL2(ctr, ...) \
    Auto_INTERNAL1(TOKEN_PASTE(Auto_func_, ctr), \
                   TOKEN_PASTE(Auto_instance_, ctr), __VA_ARGS__)

#define Auto(...) Auto_INTERNAL2(__COUNTER__, __VA_ARGS__)
```

Choose Your Own Digression

- Variadic macros and `__VA_ARGS__` (C++11)
- Token pasting and `##`
- Templates
- Lambdas (C++11)
- `__COUNTER__` (non-standard)
- `#pragma once` (non-standard)
- Style point: Aren't macros evil or something?
- Style point: Why lambdas instead of `std::function`?

__COUNTER__

It gives a new integer value every time it’s expanded.

It’s non-standard,
but every compiler in the world supports it.

I almost said ***almost*** *every compiler*,
but I can’t name any compilers that don’t support it.

We would avoid it if there were any standard way to get its functionality.
`__LINE__` kinda works, I guess...

#pragma once

#pragma once is the clearest, most efficient way to make a file idempotent.

It's non-standard,
but every compiler in the world supports it.

#ifndef, #define, and #endif have their own uses, but you don't need them (and therefore shouldn't use them) to make a file idempotent.

Idempotence

A function $f : D \rightarrow D$ is idempotent if

$$f(fx) = fx \quad \text{for all } x \text{ in } D.$$

I.e., repeated applications have the same effect as one.

(FOLDOC)

Why not std::function?

```
template <class Lambda> class AtScopeExit {
  Lambda& m_lambda;
public:
  AtScopeExit(Lambda& action) : m_lambda(action) {}
  ~AtScopeExit() { m_lambda(); }
};

AtScopeExit<decltype(lname)> aname(lname);
```

green text: what we wrote

```
class AtScopeExit {
  std::function<void(void)> m_lambda;
public:
  template<class Lambda> AtScopeExit(Lambda& action) : m_lambda(action) {}
  ~AtScopeExit() { m_lambda(); }
};

AtScopeExit aname(lname);
```

red text: what we consciously chose not to write

Why not std::function?

- We don’t want to pull in all of <functional>.
  - “auto.h” is included by generated code and must be lightweight.
- std::function uses type erasure, which uses heap allocation.
  - More on this later.
- Empirically, we get better code this way.
  - Assembly listings on next page.

Let’s see some codegen!

```
#include <stdio.h>
#include "auto.h"

extern void foo();

int main() {
  if (true) {
    Auto(puts("two"));
    puts("one");           // compiler knows this doesn't throw
  }
  if (true) {
    Auto(puts("three"));
    foo();                 // might throw an exception
  }
}
```

Let's see some codegen!

Clang 3.4 -O2 gives perfect code

```
#include <stdio.h>
#include "auto.h"

extern void foo();

int main() {
  if (true) {
    Auto(puts("two"));
    puts("one");
  }
  if (true) {
    Auto(puts("three"));
    foo();
  }
}
```

```
_main:
        pushq %rbp
        movq  %rsp, %rbp
        pushq %rbx
        pushq %rax
        leaq  L_.str(%rip), %rdi  ## "one"
        callq _puts
        leaq  L_.str2(%rip), %rdi  ## "two"
        callq _puts
        callq __Z3foov
## reached iff foo doesn't throw any exception
        leaq  L_.str1(%rip), %rdi  ## "three"
        callq _puts
        xorl  %eax, %eax
        addq  $8, %rsp
        popq  %rbx
        popq  %rbp
        ret
LBB0_2:  ## reached iff foo throws an exception
        movq  %rax, %rbx
        leaq  L_.str1(%rip), %rax  ## "three"
        movq  %rax, %rdi
        callq _puts
        movq  %rbx, %rdi
        callq __Unwind_Resume
```

To remove this stack frame, use -O3.

Let's see some codegen!

GCC 4.8 -O2 gives perfect code
(but only if you give it a hint)

```
extern void puts(const char*)
   noexcept(true);

extern void foo();

int main() {
  if (true) {
    Auto(puts("two"));
    puts("one");
  }
  if (true) {
    Auto(puts("three"));
    foo();
  }
}
```

Clang is smarter than GCC about the standard library's noexcept guarantees.

```
main:
      pushq   %rbx
      movl  $.LC0, %edi  ## "one"
      call  _Z4putsPKc
      movl  $.LC1, %edi  ## "two"
      call  _Z4putsPKc
      call  _Z3foov
## reached iff foo doesn't throw any exception
      movl  $.LC2, %edi  ## "three"
      call  _Z4putsPKc
      xorl  %eax, %eax
      popq  %rbx
      ret
.L3:  ## reached iff foo throws an exception
      movq  %rax, %rbx
      movl  $.LC2, %edi  ## "three"
      call  _Z4putsPKc
      movq  %rbx, %rdi
      call  _Unwind_Resume
```

Let's see some codegen!

The std::function version is objectively terrible.

```
#include <stdio.h>
#include "auto.h"

extern void foo();

int main() {
  if (true) {
    Auto(puts("two"
    puts("one");
  }
  if (true) {
    Auto(puts("thre
    foo();
  }
}
```

```
_main:
        pushq %rbp
        movq  %rsp, %rbp
        pushq %r14
        pushq %rbx
        subq  $64, %rsp
        movq  ___stack_chk_guard@GOTPCREL(%
rip), %r14
        movq  (%r14), %rax
        movq  %rax, -24(%rbp)
        leaq  -80(%rbp), %rbx
        movq  %rbx, -48(%rbp)
        leaq  __ZTVNSt3__110__function6__f
        movq  %rax, -80(%rbp)
        leaq  L_.str(%rip), %rdi
        callq _puts
        movq  %rbx, %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  %rbx, -48(%rbp)
        leaq  __ZTVNSt3__110__function6__f
        movq  %rax, -80(%rbp)
        callq __Z3foov
## BB#1:
        leaq  -80(%rbp), %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  (%r14), %rax
        cmpq  -24(%rbp), %rax
        jne   LBB0_4
## BB#2:
        xorl  %eax, %eax
        addq  $64, %rsp
        popq  %rbx
        popq  %r14
        popq  %rbp
        ret
LBB0_4:
        callq ___stack_chk_fail
LBB0_3:
        movq  %rax, %rbx
        leaq  -80(%rbp), %rax
        movq  %rax, %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  %rbx, %rdi
        callq __Unwind_Resume
```

Let's see some codegen!

The std::function version is objectively terrible.

```
#include <stdio.h>
#include "auto.h"

extern void foo();

int main() {
  if (true) {
    Auto(puts("two"
    puts("one");
  }
  if (true) {
    Auto(puts("thre
    foo();
  }
}
```

```
_main:
        pushq %rbp
        movq  %rsp, %rbp
        pushq %r14
        pushq %rbx
        subq  $64, %rsp
        movq  ___stack_chk_guard@GOTPCREL(%
rip), %r14
        movq  (%r14), %rax
        movq  %rax, -24(%rbp)
        leaq  -80(%rbp), %rbx
        movq  %rbx, -48(%rbp)
        leaq  __ZTVNSt3__110__function6__f
        movq  %rax, -80(%rbp)
        leaq  L_.str(%rip), %rdi
        callq _puts
        movq  %rbx, %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  %rbx, -48(%rbp)
        leaq  __ZTVNSt3__110__function6__f
        movq  %rax, -80(%rbp)
        callq __Z3foov
```

(700 lines of std::function code omitted)

```
## BB#1:
        leaq  -80(%rbp), %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  (%r14), %rax
        cmpq  -24(%rbp), %rax
        jne   LBB0_4
## BB#2:
        xorl  %eax, %eax
        addq  $64, %rsp
        popq  %rbx
        popq  %r14
        popq  %rbp
        ret
LBB0_4:
        callq ___stack_chk_fail
LBB0_3:
        movq  %rax, %rbx
        leaq  -80(%rbp), %rax
        movq  %rax, %rdi
        callq __ZN15AtScopeExitD2Ev
        movq  %rbx, %rdi
        callq __Unwind_Resume
```

So how is std::function implemented, to get such bad performance?

Type erasure in a nutshell

To capture any type:

(1) Make a Container that can hold any type.

I.e., make a template class.

```
template<typename T> class Container
{
    T captured_object;
}
```

Type erasure in a nutshell

To capture any type:

(2) Make a TypeErasedObject that can hold Container<T> for any T.

Via polymorphism (inheritance and virtual dispatch).

```
template <typename T> class Container : ContainerBase;

class TypeErasedObject {
    ContainerBase *container;
    TypeErasedObject(X x) { container = new Container<X>(x); }
};
```

#include "function.h"

```
#pragma once
#include <utility>

struct ContainerBase {
  virtual void perform() = 0;
  virtual ~ContainerBase() = default;
};

template <class Lambda> struct Container : ContainerBase {
  Lambda m_lambda;
  Container(Lambda&& lambda) : m_lambda(std::move(lambda)) {}
  virtual void perform() { m_lambda(); }
};

class function {  // equivalent to std::function<void(void)>
 ContainerBase *m_ctr;
public:
  template<class Lambda> function(Lambda lambda)
    : m_ctr(new Container<Lambda>(std::move(lambda))) {}
  void operator()() { m_ctr->perform(); }
  ~function() { delete m_ctr; }
};
```

#include "function.h"

```
#pragma once
#include <utility>

struct ContainerBase {
  virtual void perform() = 0;
  virtual ~ContainerBase() = default;
};

template <class Lambda> struct Container : ContainerBase {
  Lambda m_lambda;
  Container(Lambda&& lambda) : m_lambda(std::move(lambda)) {}
  virtual void perform() { m_lambda(); }
};

class function {  // equivalent to std::function<void(void)>
 ContainerBase *m_ctr;
public:
  template<class Lambda> function(Lambda lambda)
    : m_ctr(new Container<Lambda>(std::move(lambda))) {}
  void operator()() { m_ctr->perform(); }
  ~function() { delete m_ctr; }
};
```

std::move has a compile-time cost, as it relies on std::remove_reference

virtual dispatch has a runtime cost

we cannot avoid move-constructing a Lambda here; this move-constructs all its captures (but in our case this is cheap, because we captured them by reference)

memory allocation has a huge runtime cost, although we may avoid it if sizeof (Lambda) is small (via a kind of “small string optimization”)

Alternative syntaxes

- Alexandrescu & Marginean's ScopeGuard
- Boost.ScopeExit
- Google scope-exit

Alexandrescu & Marginean

Generic: Change the Way You Write Exception-Safe Code — Forever

Andrei Alexandrescu and Petru Marginean, December 2000

http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758

```
ScopeGuard guard = MakeObjGuard(state, &State::EnableLogging);

ON_BLOCK_EXIT(state, &State::EnableLogging);
```

Alexandrescu & Marginean

Generic: Change the Way You Write Exception-Safe Code — Forever

Andrei Alexandrescu and Petru Marginean, December 2000

http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758

```
ScopeGuard guard = MakeObjGuard(state, &State::EnableLogging);

ON_BLOCK_EXIT(state, &State::EnableLogging);
```

Can’t run arbitrary code unless it’s wrapped in a function

Can’t write your cleanup code in-line with your other code

Cleanup code can’t refer to local variables

Boost.ScopeExit

Plain vanilla Boost:

```
    BOOST_SCOPE_EXIT(&state) {
        state->EnableLogging();
    } BOOST_SCOPE_EXIT_END
```

Or, if you have C++11, Boost provides:

```
    BOOST_SCOPE_EXIT_ALL(&) { state->EnableLogging(); };
```

Or, a C++11 alternative suggested in the Annex:

```
    scope_exit on_exit42([&]{ state->EnableLogging(); });
```

Boost.ScopeExit

Plain vanilla Boost:

```
    BOOST_SCOPE_EXIT(&state) {
        state->EnableLogging();
    } BOOST_SCOPE_EXIT_END
```

Or, if you have C++11, Boost provides:

```
    BOOST_SCOPE_EXIT_ALL(&) { state->EnableLogging(); };
```

Or, a C++11 alternative suggested in the Annex:

```
    scope_exit on_exit42([&]{ state->EnableLogging(); });
```

Very similar to Auto(), but so much boilerplate!

scope_exit requires coming up with unique names (not friendly to code-generation)

Google scope-exit

```
    ON_SCOPE_EXIT((state), state->EnableLogging());
```

An example from their documentation:

```
    template<typename T>
    void f(T& t)
    {
        int i, x;

        ON_SCOPE_EXIT((i) SCOPE_EXIT_TEMPLATE_VAR(t) (x),
            /* Do something with i, t, and x */
        );
    }
```

Google scope-exit

```
    ON_SCOPE_EXIT((state), state->EnableLogging());
```

An example from their documentation:

```
    template<typename T>
    void f(T& t)
    {
        int i, x;

        ON_SCOPE_EXIT((i) SCOPE_EXIT_TEMPLATE_VAR(t) (x),
            /* Do something with i, t, and x */
        );
    }
```

Must explicitly name all your captures (unfriendly to code-generation)

Weird corner cases with templates and the “this” pointer

One more time

```
#pragma once

template <class Lambda> class AtScopeExit {
  Lambda& m_lambda;
public:
  AtScopeExit(Lambda& action) : m_lambda(action) {}
  ~AtScopeExit() { m_lambda(); }
};

#define TOKEN_PASTEx(x, y) x ## y
#define TOKEN_PASTE(x, y) TOKEN_PASTEx(x, y)

#define Auto_INTERNAL1(lname, aname, ...) \
    auto lname = [&]() { __VA_ARGS__; }; \
    AtScopeExit<decltype(lname)> aname(lname);

#define Auto_INTERNAL2(ctr, ...) \
    Auto_INTERNAL1(TOKEN_PASTE(Auto_func_, ctr), \
                   TOKEN_PASTE(Auto_instance_, ctr), __VA_ARGS__)

#define Auto(...) Auto_INTERNAL2(__COUNTER__, __VA_ARGS__)
```

One odd application

```
CodePrinter& code = context.codeprinter;

code.Printf("void MergeWith(OtherRowElement* other, const TableColumns_%s*
/*dummy*/, int threadId)\n", ti[i].tableAlias);
code.Scope();
code.Printf("if (other->%s == nullptr)\n", ti[i].tableResultName);
code.Scope();
code.Printf("%s = nullptr;\n", ti[i].tableResultName);
code.Unscope();
code.Printf("else\n");
code.Scope();
CodeGenElseBlock(context, ti, i);
code.Unscope();
code.Unscope();  // end of function body
```

One odd application

```
#define AutoScope(code) code.Scope(); Auto(code.Unscope());

code.Printf("void MergeWith(OtherRowElement* other, const TableColumns_%s*
/*dummy*/, int threadId)\n", ti[i].tableAlias);
{
    AutoScope(code);
    code.Printf("if (other->%s == nullptr)\n", ti[i].tableResultName);
    {
        AutoScope(code);
        code.Printf("%s = nullptr;\n", ti[i].tableResultName);
    }
    code.Printf("else\n");
    {
        AutoScope(code);
        CodeGenElseBlock(context, ti, i);
    }
}
```

Any questions?

```
#pragma once

template <class Lambda> class AtScopeExit {
  Lambda& m_lambda;
public:
  AtScopeExit(Lambda& action) : m_lambda(action) {}
  ~AtScopeExit() { m_lambda(); }
};

#define TOKEN_PASTEx(x, y) x ## y
#define TOKEN_PASTE(x, y) TOKEN_PASTEx(x, y)

#define Auto_INTERNAL1(lname, aname, ...) \
    auto lname = [&]() { __VA_ARGS__; }; \
    AtScopeExit<decltype(lname)> aname(lname);

#define Auto_INTERNAL2(ctr, ...) \
    Auto_INTERNAL1(TOKEN_PASTE(Auto_func_, ctr), \
                   TOKEN_PASTE(Auto_instance_, ctr), __VA_ARGS__)

#define Auto(...) Auto_INTERNAL2(__COUNTER__, __VA_ARGS__)
```

Chapter 2. make_iterable

Inside-out containers

What we start with

```
class MDTable
{
  MDColumn *m_columns;
  MDKey *m_keys;
  int m_columnCount;
  int m_keyCount;
public:
  MDColumn* GetColumns() const { return m_columns; }
  int GetNumColumns() const { return m_columnCount; }
  MDIndex* GetKeys() const { return m_keys; }
  int GetNumKeys() const { return m_keyCount; }
};
```

Why not just use std::vector?

```
class MDTable
{
  MDKey *m_keys;  // both normal and foreign
  int m_keyCount;
  int m_firstForeignKey;

public:
  MDIndex* GetNormalKeys() const { return m_keys; }
  int GetNumNormalKeys() const { return m_firstForeignKey; }
  MDIndex* GetForeignKeys() const { return m_keys + ...; }
  int GetNumForeignKeys() const { return m_keyCount - ...; }
};
```

Why not just use std::vector?

```
class MDTable
{
  MDKey *m_keys;  // both normal and fo
  int m_keyCount;
  int m_firstForeignKey;

public:
  MDIndex* GetNormalKeys() const { return m_keys; }
  int GetNumNormalKeys() const { return m_firstForeignKey; }
  MDIndex* GetForeignKeys() const { return m_keys + ...; }
  int GetNumForeignKeys() const { return m_keyCount - ...; }
};
```

Weird design choices.

Time efficiency.

Space efficiency.

Using GetColumns() is cumbersome

```
void TransformTable(MDTable *tab)
{
  for (int i=0; i < tab->GetNumColumns(); ++i)
  {
    MDColumn& col = tab->GetColumns()[i];
    ... col ...
  }
  for (int i=0; i < tab->GetNumKeys(); ++i)
  {
    MDKey& key = tab->GetKeys()[i];
    ... key ...
  }
}
```

What we'd like to write

```
void TransformTable(MDTable *tab)
{
  for (MDColumn& col : Columns(tab))
  {
    ... col ...
  }
  for (MDKey& key : Keys(tab))
  {
    ... key ...
  }
}
```

Our Columns() and Keys() functions

```
#include "iterable.h"

static inline iterable<MDColumn*> Columns(MDTable* tab)
{
  MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}

static inline iterable<MDKey*> Keys(MDTable* tab)
{
  MDKey* keys = tab->GetKeys();
  return make_iterable(keys, keys + tab->GetNumKeys());
}
```

Our Columns() and Keys() functions

```
#include "iterable.h"

static inline iterable<MDColumn*> Columns(MDTable* tab)
{
  MDColumn* cols = tab->GetColumns();
  return make_iterable(cols, cols + tab->GetNumColumns());
}

static inline iterable<MDKey*> Keys(MDTable* tab)
{
  MDKey* keys = tab->GetKeys();
  return make_iterable(keys, keys + tab->GetNumKeys());
}
```

#include "iterable.h"

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};


template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

#include "iterable.h"

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};

template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

Marshall Clow called it **`iterator_pair`**

http://cplusplusmusings.wordpress.com/2013/04/14/range-based-for-loops-and-pairs-of-iterators/

Google calls it **`std::range`**

http://cxx1y-range.googlecode.com/git/paper.html

Boost calls it **`iterator_range`**

http://www.boost.org/doc/libs/1_53_0/libs/range/doc/html/range/reference/utilities/iterator_range.html

Alisdair Meredith (N2977) called it **`std::range`**

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2977.pdf

It's basically just a pair of iterators...

Original Frankfurt C++11 proposal called it `std::pair`

This is a summary of Alisdair Meredith's N2977, "Pairs do not make good ranges."

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

This is a summary of Alisdair Meredith's N2977, "Pairs do not make good ranges."

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

because there are standard algorithms
that deal in pairs of iterators
that are **not ranges**

This is a summary of Alisdair Meredith's N2977, "Pairs do not make good ranges."

Original Frankfurt C++11 proposal called it `std::pair`

but this is a bad idea

because there are standard algorithms

that deal in pairs of iterators

that are **not ranges**

```
template <class InputIt1, class InputIt2>
std::pair<InputIt1,InputIt2> mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2);

template<class ForwardIt>
std::pair<ForwardIt, ForwardIt> minmax_element(ForwardIt first, ForwardIt last);

template<class InputIt, class OutputIt1, class OutputIt2, class UnaryPredicate>
std::pair<OutputIt1, OutputIt2> partition_copy(InputIt first, InputIt last,
                                               OutputIt1 d_first_true, OutputIt2 d_first_false,
                                               UnaryPredicate p);
```

Inside-out containers

```
template<class It>
class iterable
{
  It m_first, m_last;
public:
  iterable() = default;
  iterable(It first, It last) :
    m_first(first), m_last(last) {}
  It begin() const { return m_first; }
  It end() const { return m_last; }
};

template<class It>
inline iterable<It> make_iterable(It a, It b)
{
  return iterable<It>(a, b);
}
```

Make a “container view” of an object on the fly

One object can have multiple iterable parts, without exposing implementation details

You can iterate over a subrange as easily as you iterate over the entire container

Still no word on “ranges” in C++1z (there is a working group)

Chapter 3. 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`

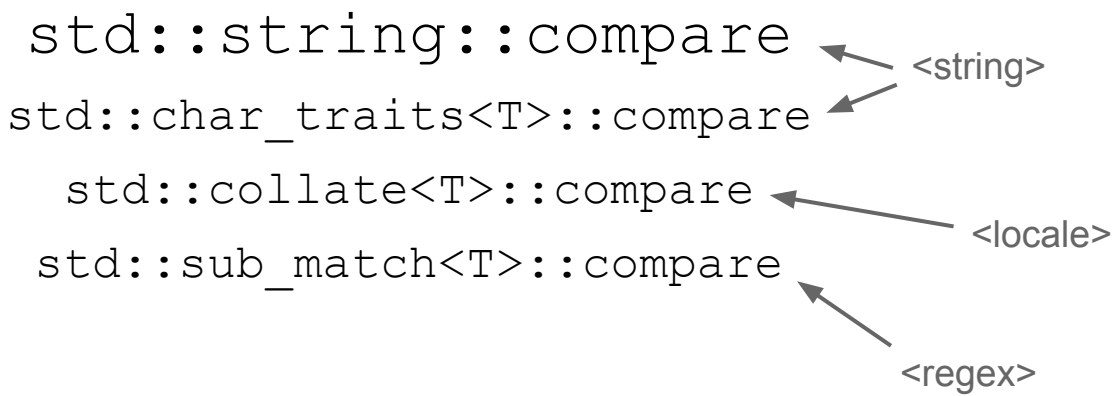
(`strcoll`, `strcasecmp`...)

`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
```

This won't be about operator overloading

We don't care how the operation is spelled

(and spelling it `<=>` seems out of the question)

We can just spell it **`std::spaceship`**.

That's fine.

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 implement 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 implement it (library style)

```
// 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 implement 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 implement it (library style)

```
// 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 (libc++ style)

```
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 hard part (Painless Metaprogramming style)

```
template <class _Tp, class _Up, size_t ..._Ip>
constexpr int __tuple_spaceship(const _Tp& __x, const _Up& __y,
                                const index_sequence<_Ip...>&)
{
    int __r = 0;
    std::initializer_list<int> x = {
        (__r != 0) ? 0 : (__r = spaceship(get<_Ip>(__x), get<_Ip>(__y))) ...
    };
    return __r;
}

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(__x, __y, make_index_sequence<sizeof...(_Tp)>{});
}
```

Clang 3.4 doesn’t seem to care which way you do it.

Recommended reading: “Towards Painless Metaprogramming” http://ldionne.github.io/mpl11-cppnow-2014/

The idiom we *should* use in C++1z

```
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++1z

```
/// 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++1z.

You know anyone on the standards committee?

Postscript

```
struct S
{
    void operator<= (int) {}
};

template<void (S::*)(int)> void f() {}

int main()
{
    f<&S::operator<=>();
}
```

```
end()
```