# 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__)
```