Efficient Modern C++

Marco Foco – C++ User Group Udine 2016/12/03
Performance in Modern C++

Who am I?

- Marco Foco
  - C++ developer since 1995
  - Working at NVIDIA/Gameworks since October 2015
    - Deep learning engineer, Library designer
  - Past experience
    - Signal & audio/video processing applications
    - Machine learning
    - Automotive
    - Embedded applications
  - Blog: http://marcofoco.com, Twitter: @marcofoco

Efficient Modern C++

In the last episode...

override

```
struct A {
    virtual int value();
};

class B : public A {
    float value() override; // error!
};
```

final

```
struct A {
    virtual int value();
};

struct B : public A {
    int value() final;
};

struct C : public B {
    int value();        // error!
};
```

Is that all?

- Modern C++ offers more than that!
- Coming up next:
  - Devirtualization
  - Move semantics
  - Passing functions

Efficient Modern C++

Devirtualization

Devirtualization

- Compile-time replacement of a virtual function.
- Enabled by
  - static deduction of a specific type
  - whole program optimizations (maybe?)
  - giving the compiler hints

Devirtualization (example)

```
struct A {
    virtual int value() { return 1; }
};

struct B : public A {
  int value() { return 2; }
};

int test(B* b) { return b->value() + 11; }
```

Static type deduction

- If the use of our function test is something like:

```
B b;
test(&b);
```

- the compiler can deduce that this call to `test` happens with a B
- The body of test is optimized as it contained a simple `return 13;`

Global optimizations (theory)

- The compiler can deduce there are no subclasses of `B` overriding `B::value()`.
- In such case, the function call in test() could be a good candidate for devirtualization.

Module 1

```
struct A {
    virtual int value() { return 1; }
};

struct B : public A {
    int value() { return 2; }
};

int test(B* b) { return b->value() + 11; }
```

Module 2

```
struct C : public B {
    int other;
    C() : other(5) {}
};
```

Module 3

```
int main() {
    ...
    B *b = ...; // input-dependent expr
    test(b);    // devirtualization?
}
```

Global optimizations (practice)

- No compiler will ever devirtualize those calls
- Dynamic linking could add new classes!
- Module 4 (dynamically added at runtime)

```
struct D : public B {
    int value() { return 3; }
};
```

- Global optimizations can be only used for finding more static type deductions, and optimizing those calls.

Devirtualization via hints

- We've seen that C++ 11 introduces the keyword final to seal a method

```
struct B : public A {
    int value() final { return 2; }
};
```

- or a class

```
struct B final : public A {
    int value() { return 2; }
};
```

- In both cases the compiler is able to deduce that the method only `B::value()` will ever be called.

Hints: results in VS2013+*

- Without final specifier

```
sub rsp, 40
mov rax, qword ptr [rcx]
call qword ptr [rax]
add eax, 11
add rsp, 40
ret
```

- With final specifier

```
mov eax, 13
ret
```

Hints: results in clang 3.0+

- Without final specifier

```
push  rax
mov rax, qword ptr [rdi]
call qword ptr [rax]
add eax, 11
pop rcx
ret
```

- With final specifier

```
mov eax, 13
ret
```

Hints: results in GCC 4.7+

- Without final specifier

```
    mov rax, qword ptr [rdi]
    mov rdx, qword ptr [rax]
    cmp rdx, offset B::value
    jne .L12
    mov eax, 13
    ret
.L12:
    sub rsp, 8
    call rdx
    add rsp, 8
    add eax, 11
    ret
```

- With final specifier

```
mov eax, 13
ret
```


Partial devirtualization!

Devirtualization: Conclusions

- It's NOT a mandatory optimization
- It's performed by most recent compilers with -O2
  - GCC >= 4.7 (GCC also applies it with just -O1)
  - Clang >= 3.2
  - Visual Studio >= 2013
- final also improves safety: **win-win!**

Efficient Modern C++

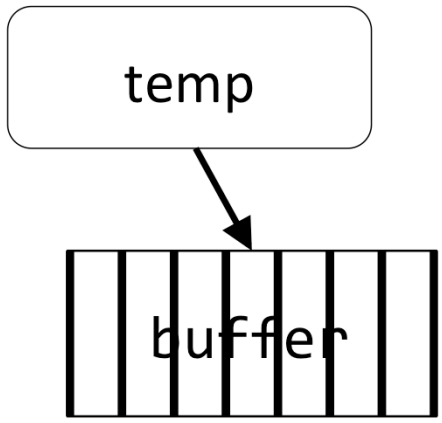
move semantics

Example: Factory Pattern

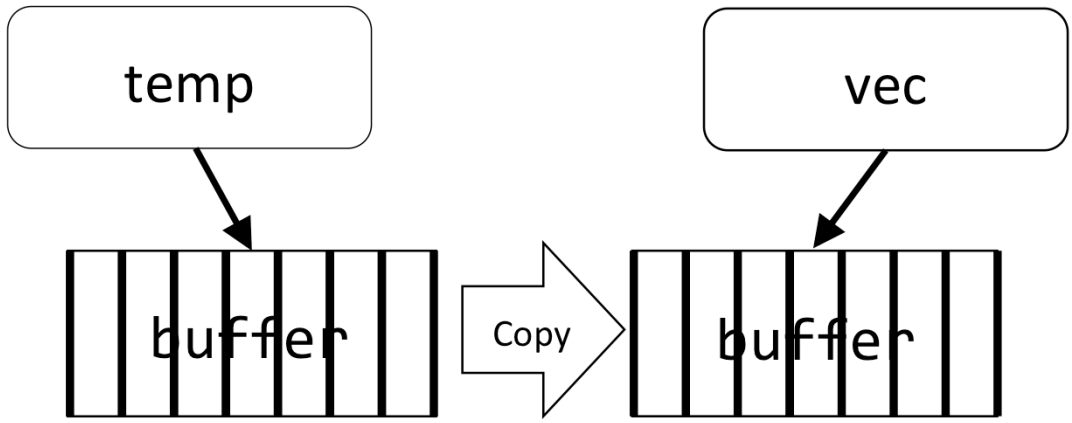
```
std::vector<float> VectorFactory(int size)
{
     std::vector<float> temp(size, 0);
// do something
     return temp;
}

std::vector<float> vec =
VectorFactory(10);
```

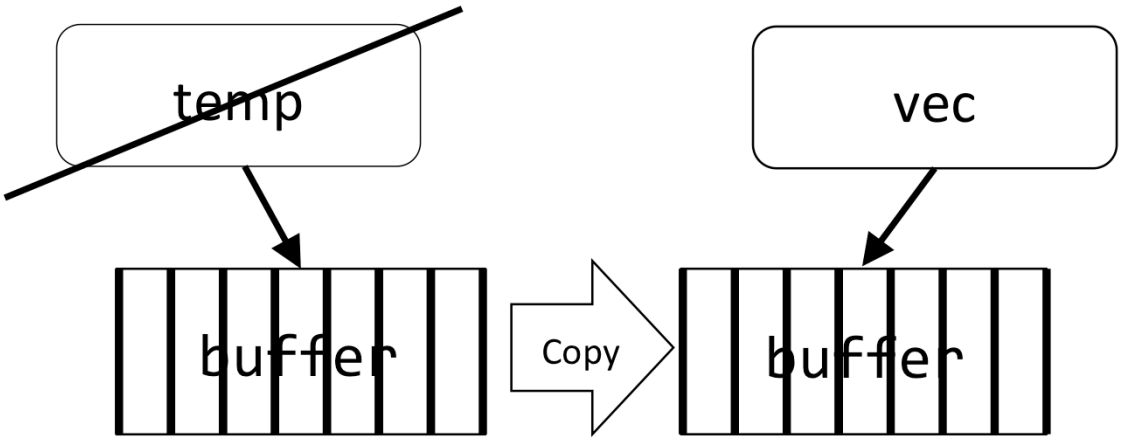
Copy semantics



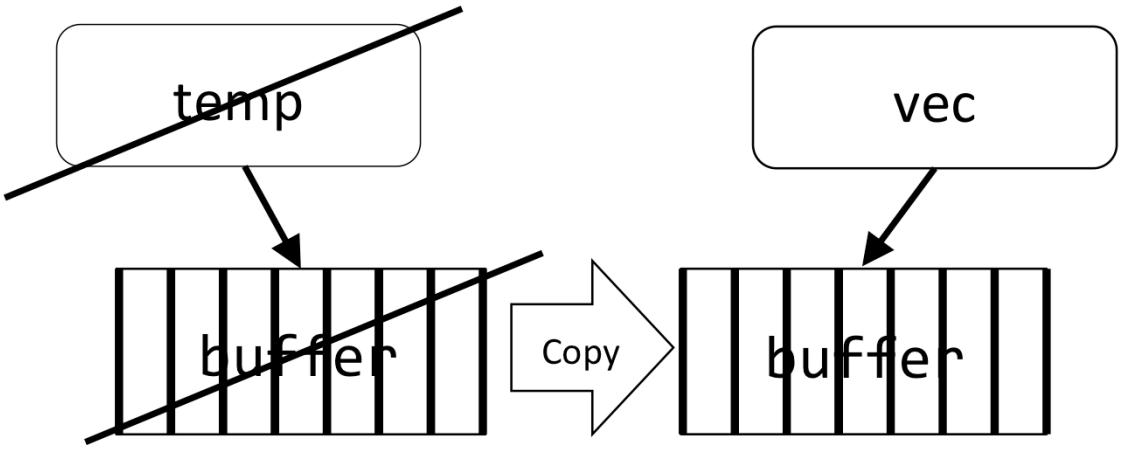
Copy semantics



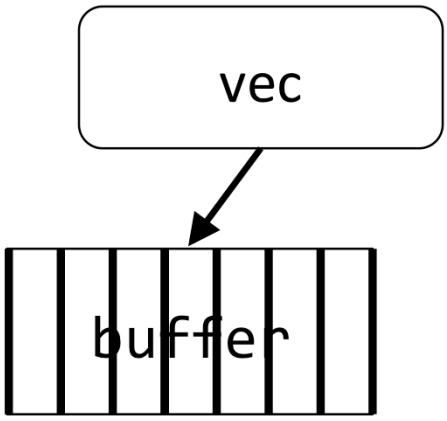
Copy semantics



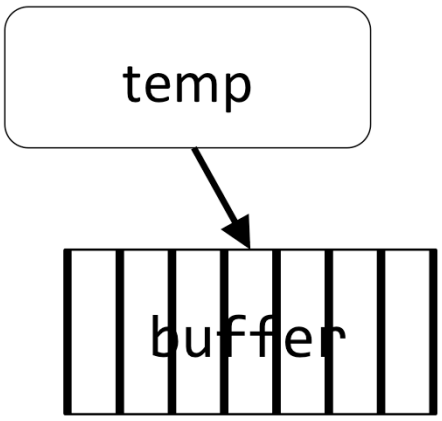
Copy semantics



Copy semantics

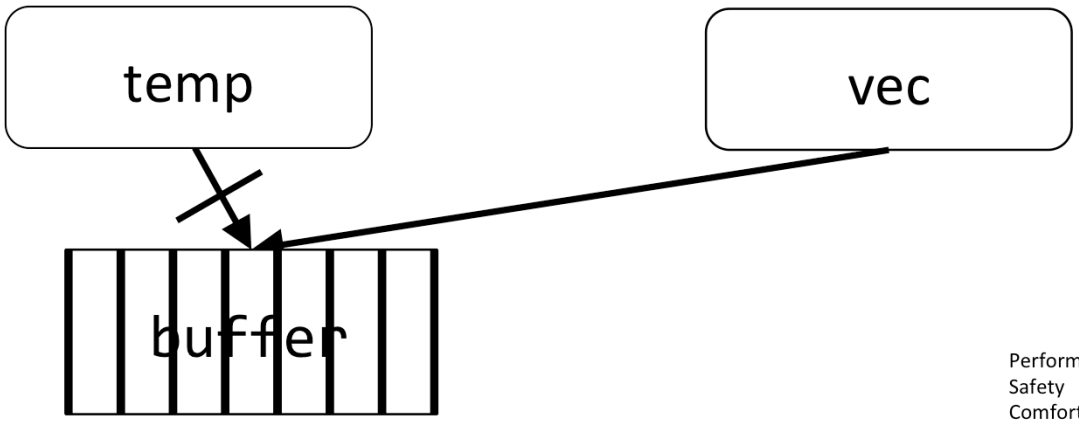


Move semantics



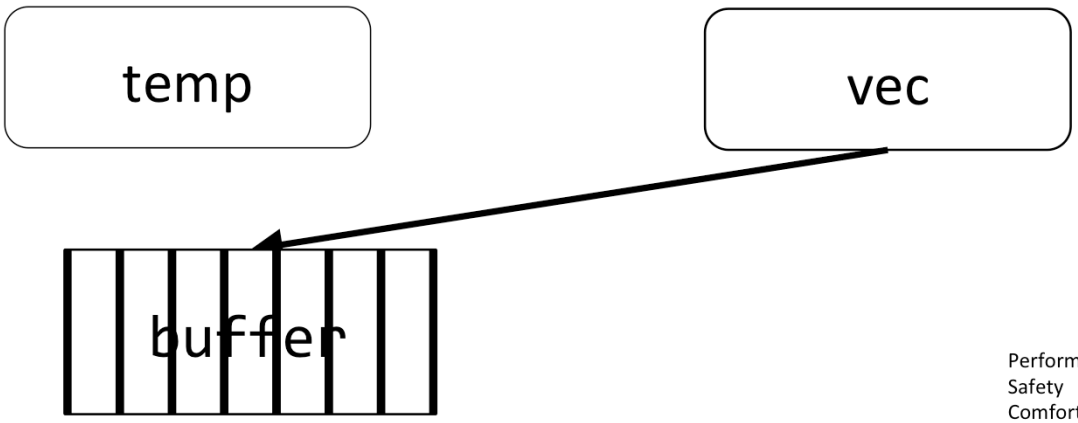
Performance
Safety
Comfort

Move semantics



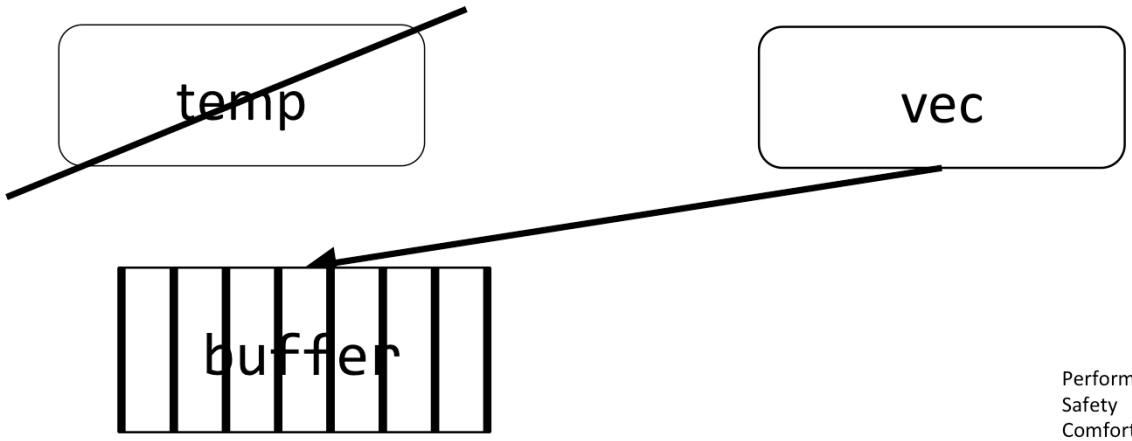
Performance
Safety
Comfort

Move semantics



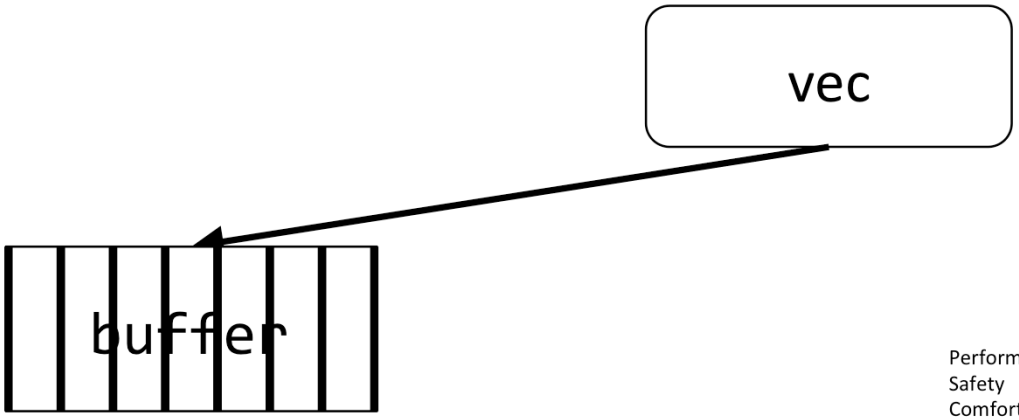
Performance
Safety
Comfort

Move semantics



Performance
Safety
Comfort

Move semantics



Performance
Safety
Comfort

Compiler generated methods

Compiler generates (if possible)

Declare	Default ctor	Copy ctor	Copy assign	Move ctor	Move assign	dtor
Any ctor	no	yes	yes	yes	yes	yes
Copy ctor	yes		yes*	no	no	yes
Copy assign	yes	yes*		no	no	yes
Move ctor	yes	no	no		no	yes
Move assign	yes	no	no	no		yes
dtor	yes*	yes*	yes*	no	no	

yes* should be no (backwards compatible)

Note: when you do not write any ctors, everything gets generated (if possible)

How to get movability?

- Resource-managing classes
  - Apply the rule of 5: write your own move ctor, move assign

```
Widget(Widget&&);
Widget& operator=(Widget&&);
```

  - Also explicitly delete unwanted constructors
- For other classes, get movability for free
  - Apply the Rule of 0

Free movability

- Basic encapsulation in pre-C++11

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const { return _s; }
};
```

- In Modern C++ receives an upgrade: free movability

Static factories

- Automatically created move constructor and assignment for class `C` exploit its content's movability

```
C c1 = ...;
...
C c2 = std::move(c1); // this won't copy c1._s
```

- It’s now possible, and convenient, to use static factories:

```
C f() { return C("test"); }
```

Examples

- Direct Use

```
cout << f().get();
```

- Store C

```
C c = f();
cout << c.get();
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store C by val and string by ref

```
C c = f();
const string &s = c.get();
cout << s;
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store C by ref

```
const C &c = f();
cout << c.get();
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store C by ref

```
const C &c = f();
cout << c.get();
```

- Isn't c a dangling reference?!?

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store C by ref

```
const C &c = f();
cout << c.get();
```

- Isn't c a dangling reference?!?
  - Nope
  - Temporary Lifetime Extension!

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Temporary Lifetime Extension

- These two pieces of code are equivalent:

```
const C c = f();
cout << c.get();
```

```
const C &c = f();
cout << c.get();
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store C by ref and string by ref

```
const C &c = f();
const string &s = c.get();
cout << s;
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store the string by ref

```
const string &s = f().get();
cout << s;
```

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

Examples

- Store the string by ref

```
const string &s = f().get();
cout << s;
```

FAIL

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const {
        return _s;
    }
};

C f() { return C("test"); }
```

We have the technology…

```
class C {
    string _s;
public:
    C(const std::string &s) : _s(s) {}
    const string &get() const & { return _s; }
    string get() && { return move(_s); }
};
```

- The r-value reference overload will be used on temporaries
- It will move the value of _s away from the temporary object.

r-value reference overloads

- Store the string by ref

```
const string &s = f().get();
cout << s;
```

PASS

```
class C {
    string _s;
public:
    C(const string &s) : _s(s) {}
    const string &get() const & {
        return _s;
    }
    string get() && {
        return move(_s);
    }
};

C f() { return C("test"); }
```

r-value reference… at a cost!

- Incidentally, the behaviour of direct uses change as well:

```
cout << f().get();
```

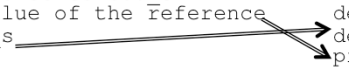
- This could lead to a loss of performance

Before

```
creating C
creating C::_s
moving C (return of f())
getting a reference to C::_s
printing the value of the reference
destroying C::_s
destroying C
```

After

```
creating C
creating C::_s
moving C (return of f())
moving C::_s into temporary
destroying C::_s
destroying C
printing the value of temporary
```



Small performance loss. Big performance loss if parts of the object are non-movable (but copyable)

Efficient Modern C++

Passing functions

Passing Functions

- C++11 introduces new ways to create functions
  - Lambda
    - Stateless
    - Stateful
- and an additional method to encapsulate them
  - `std::function<...>`

Stateless lambda

- Stateless lambdas automatically convert to function pointers

```
using F = void(int);
void apply_stateless(int* b, int* e, F* f) {
    while (b != e) {
        f(*b);
        ++b;
    }
}
...
apply_stateless(v, v + n, [](int i) { cout << i; });
```

Stateless lambda

- There's no state in stateless lambdas
  - must use globals

```
static int g_acc; // global

int test_stateless(int *v, int n) {
    g_acc = 0;
    apply_stateless(v, v + n, [](int i) { g_acc += i; });
    return g_acc;
}
```

Stateless lambda: performance

- After inlining, the result is rather optimized (GCC 4.7+, Clang 3.3+):

```
test_stateless(int*, int):
        movsx   rsi, esi
        xor     eax, eax
        mov     dword ptr g_acc, 0
        lea     rdx, [rdi+rsi*4]
        cmp     rdi, rdx
        je      .L4
.L3:
        add     eax, DWORD PTR [rdi]
        add     rdi, 4
        cmp     rdx, rdi
        jne     .L3
        mov     dword ptr g_acc, eax
.L4:
        ret
```

Stateful lambdas

- Can capture variables and use them later

```
int test_stateful_generic(int *v, int n) {
    int acc = 0;
    apply_stateful(v, v + n, [&](int i) { acc += i; });
    return acc;
}
```

Stateful lambdas in pre-C++11

- This code produces exactly the same result as the stateful lambda:

```
class MyLambda {
    int& acc;
public:
    MyLambda(int& _acc) : acc(_acc) {}
    void operator() (int i) { acc += i; }
};

int test_stateful(int *v, int n) {
    int acc = 0;
    apply_stateful(v, v + n, MyLambda(acc));
    return acc;
}
```

Implementing apply_stateful

- We can use `std::function`!
  - It's conceived to wrap anything
    - functions
    - stateless lambdas
    - stateful lambdas
    - methods (via `std::bind`)
- It's its job, after all.

Implementing apply_stateful

```
void apply_stateful(int* b, int* e, const function<void(int)> &f)
{
    while (b != e) {
        f(*b);
        ++b;
    }
}
```

- Simple, elegant and...

Implementing apply_stateful

- A bit convoluted…

```
std::_Function_handler<void (int), test_s...
        mov     rax, QWORD PTR [rdi]
        mov     edx, DWORD PTR [rsi]
        add     DWORD PTR [rax], edx
        ret
std::_Function_base::_Base_manager<test_s...
        cmp     edx, 1
        je      .L4
        jb      .L5
        cmp     edx, 2
        jne     .L3
        mov     rax, QWORD PTR [rsi]
        mov     QWORD PTR [rdi], rax
.L3:
        xor     eax, eax
        ret
.L5:
        mov     QWORD PTR [rdi], OFFSET FLAT:typ...
        xor     eax, eax
        ret
.L4:
        mov     QWORD PTR [rdi], rsi
        xor     eax, eax
        ret
test_stateful_function(int*, int):
        push    r12
        push    rbp
        movsx   rsi, esi
        push    rbx
        lea     r12, [rdi+rsi*4]
        sub     rsp, 48
        cmp     rdi, r12
        lea     rax, [rsp+8]
        mov     DWORD PTR [rsp+8], 0
        mov     QWORD PTR [rsp+40], OFFS...
        mov     QWORD PTR [rsp+32], OFFS...
        mov     QWORD PTR [rsp+16], rax
        je      .L16
        mov     eax, DWORD PTR [rdi]
        mov     rbx, rdi
        lea     rbp, [rsp+16]
        mov     DWORD PTR [rsp+12], eax
        mov     eax, OFFSET FLAT:std::_F...
        jmp     .L10
.L12:
        cmp     QWORD PTR [rsp+32], 0
        mov     eax, DWORD PTR [rbx]
        mov     DWORD PTR [rsp+12], eax
        je      .L26
        mov     rax, QWORD PTR [rsp+40]
.L10:
        lea     rsi, [rsp+12]
        mov     rdi, rbp
        call    rax
        add     rbx, 4
        cmp     r12, rbx
        jne     .L12
        mov     rax, QWORD PTR [rsp+32]
        test    rax, rax
        je      .L13
.L9:
        mov     edx, 3
        mov     rsi, rbp
        mov     rdi, rbp
        call    rax
.L13:
        mov     eax, DWORD PTR [rsp+8]
        add     rsp, 48
        pop     rbx
        pop     rbp
        pop     r12
        ret
.L16:
        mov     eax, OFFSET
FLAT:std::_Function_base::_Base_manager<t...
        lea     rbp, [rsp+16]
        jmp     .L9
.L26:
        call
std::__throw_bad_function_call()
        mov     rbx, rax
        mov     rax, QWORD PTR [rsp+32]
        test    rax, rax
        je      .L15
        mov     edx, 3
        mov     rsi, rbp
        mov     rdi, rbp
        call    rax
.L15:
        mov     rdi, rbx
        call _Unwind_Resume
```

Implementing apply_stateful

- Is it efficient?
  - Nope: `std::function` example is ~4-5 times slower
- Why?
  - `std::function` uses type erasure
  - Implementation is similar to virtual calls
- Rule of thumb
  - `std::function ~ virtual`

apply_generic

- A more efficient implementation uses genericity (templates)

```
template <typename TFunc>
void apply_generic(int* b, int* e, TFunc &&f) {
    while (b != e) {
        f(*b);
        ++b;
    }
}
```

- Incidentally, this version behave exactly like apply_stateless when a function pointer or a stateless lambda are passed.

apply_generic

- After inlining, the result is rather optimized (GCC 4.7+, Clang 3.3+):

```
test_stateful_generic(int*, int):
        movsx   rsi, esi
        xor     eax, eax
        lea     rdx, [rdi+rsi*4]
        cmp     rdi, rdx
        je      .L10
.L9:
        add     eax, DWORD PTR [rdi]
        add     rdi, 4
        cmp     rdx, rdi
        jne     .L9
.L10:
        ret
```

- It's even smaller than the stateless+global version!
  If you can, pass function as templates.

Passing functions: conclusions

- Select the most appropriate way of passing functions
- Know the hidden costs

Efficient Modern C++

Conclusions

Conclusions

- Use of new language features have some impact
  - `final` enables devirtualization
  - `&&` enables move-semantics
  - `&&` overloads enable flexibility
  - lambdas improves readability
  - `std::function` flexibility comes with a non-negligible cost

Questions?