Demystifying `std::function`

Maciek Gajewski, maciej.gajewski0@gmail.com
October 2016

What is std::function

- Type-erasing container
- Able to store at most one item
- Stores callable objects (functors, function pointers)

Misconceptions

- std::function is the type of the lambdas
- std::function requires some built-it compiler magic to be implemented
- std::function is horribly slow

What is it - example

```
using Fun = std::function<double(double, double)>;

Fun f1; // default constructed
assert(!f1); // converted to bool

// will throw std::bad_function_call if called
// while empty
f1(x, y); // < -- will throw
```

Example – fn pointer

```
// Can be assigned function pointer
Fun f2 = &std::atan2;
assert(!!f2);

// when invoked, invokes the pointed-to function
double r1 = f2(x, y);
double r2 = std::atan2(x, y);

assert(r1 == r2);
```

Example - functor

```
struct CounterFtor {
    int counter_ = 0;
    int operator() () { return counter_++; }
};

using CounterFun = std::function<int()>;

CounterFun ctr1 = CounterFtor{};
CounterFun ctr2 = [c=int{}]() mutable { return c++; };
```

Example - other

```
// Can be cleared by assigning nullptr
f2 = nullptr;
assert(!f2);

// type of the stored functor can be retrieved
const std::type_info& ti = ctr1.target_type();
assert(ti == typeof(CounterFtor));
```

Let's write one!

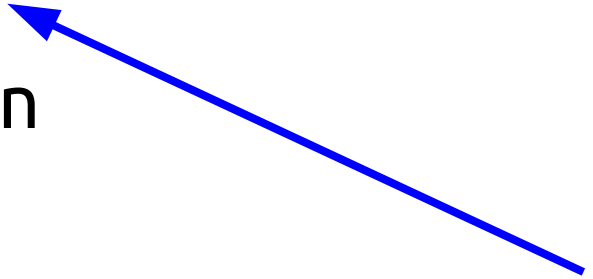
Let's write one!
But how?

Writing our own

```
template<???>
class function
{
};
```

Writing our own

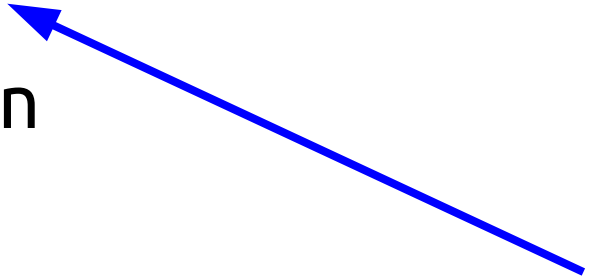
```
template<???>
class function
{
};
```



What do we put in here?

Writing our own

```
template<???>
class function
{
};
```



What do we put in here?

What can accept any of:

```
function<int()>;
function<void(int)>;
function<const double&(std::string*)>;
```

Function type

```
using FT = int(double);
```

Function type

```
using FT = int(double);

FT foo;
```

Function type

```
using FT = int(double);

FT foo; // function declaration; same as:
int foo(double);
```

Function type

```
using FT = int(double);

FT foo; // function declaration; same as:
int foo(double);

class Widget {
    FT bar;
};
```

C++ 14 standard, 8.3.5/6.3

Writing our own

```
template<typename FunType>
class function; // no definition


// partial specialization
template<typename Ret, typename Args...>
class function<Ret(Args...)>
{
    // actual definition
};
```

Writing our own

```
template<typename Ret, typename Args...>
class function<Ret(Args...)>
{
public:
   function();
   function(const function&);
   template<typename Ftor> function(Ftor f);
   Ret operator()(Args... args) const;
   operator bool () const;
// ...
};
```

Type erasure - interface

```
template<typename Ret, typename Args...>
struct functor_iface
{
   virtual ~functor_iface() = default;
   virtual functor_iface* copy() const = 0;
   virtual Ret invoke(Args... &&args) = 0;

   // ...
};
```

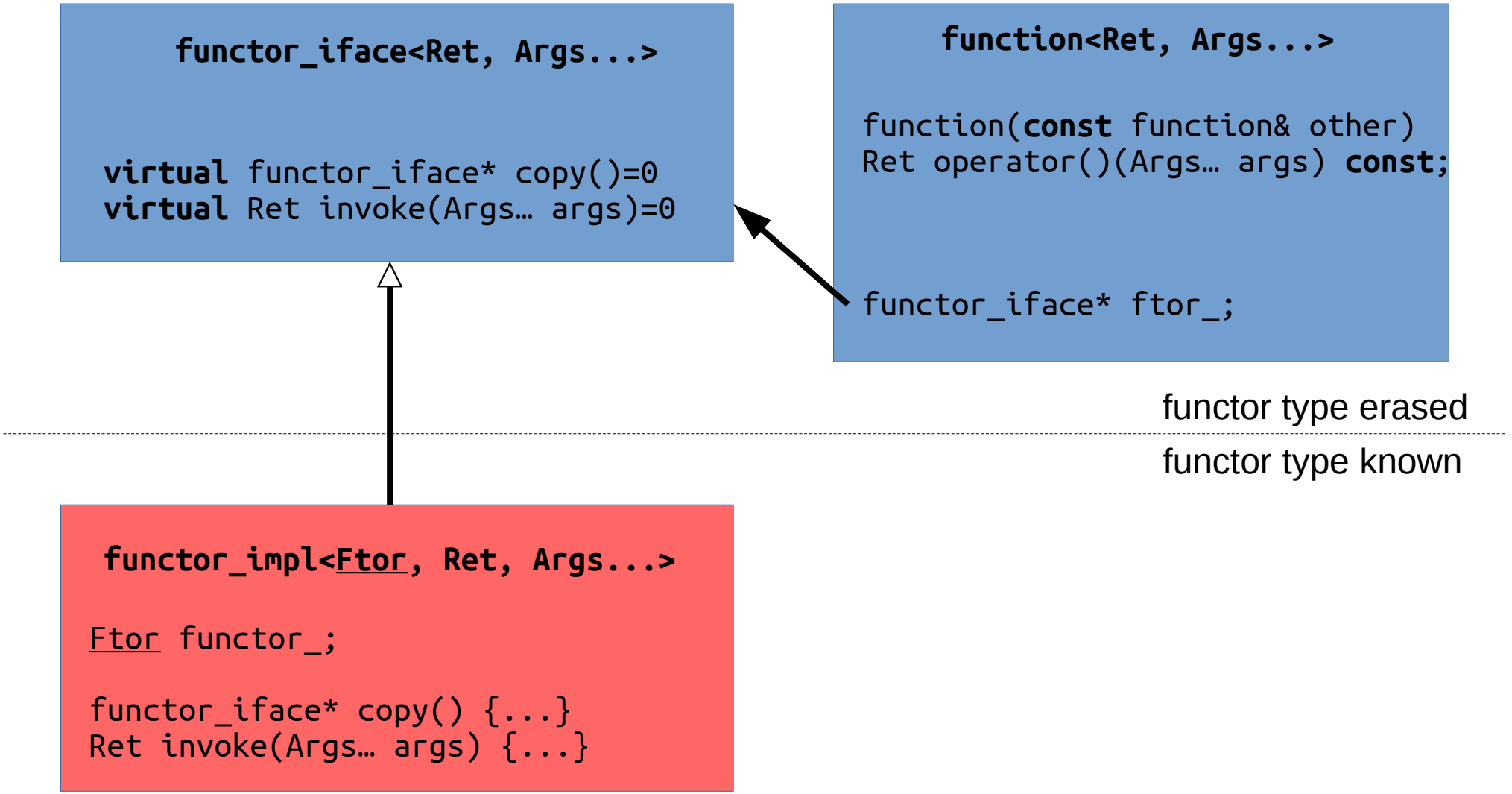
Type erasure - implementation

```
template<typename Ftor,typename Ret, typename Args...>
struct functor_impl
   : public functor_iface<Ret, Args...>
{
   Ftor functor_;
   functor_impl(const Ftor& f) : functor_(f) {}
   functor_iface* copy() const override;
   Ret invoke(Args... &&args) override;

   // ...
};
```

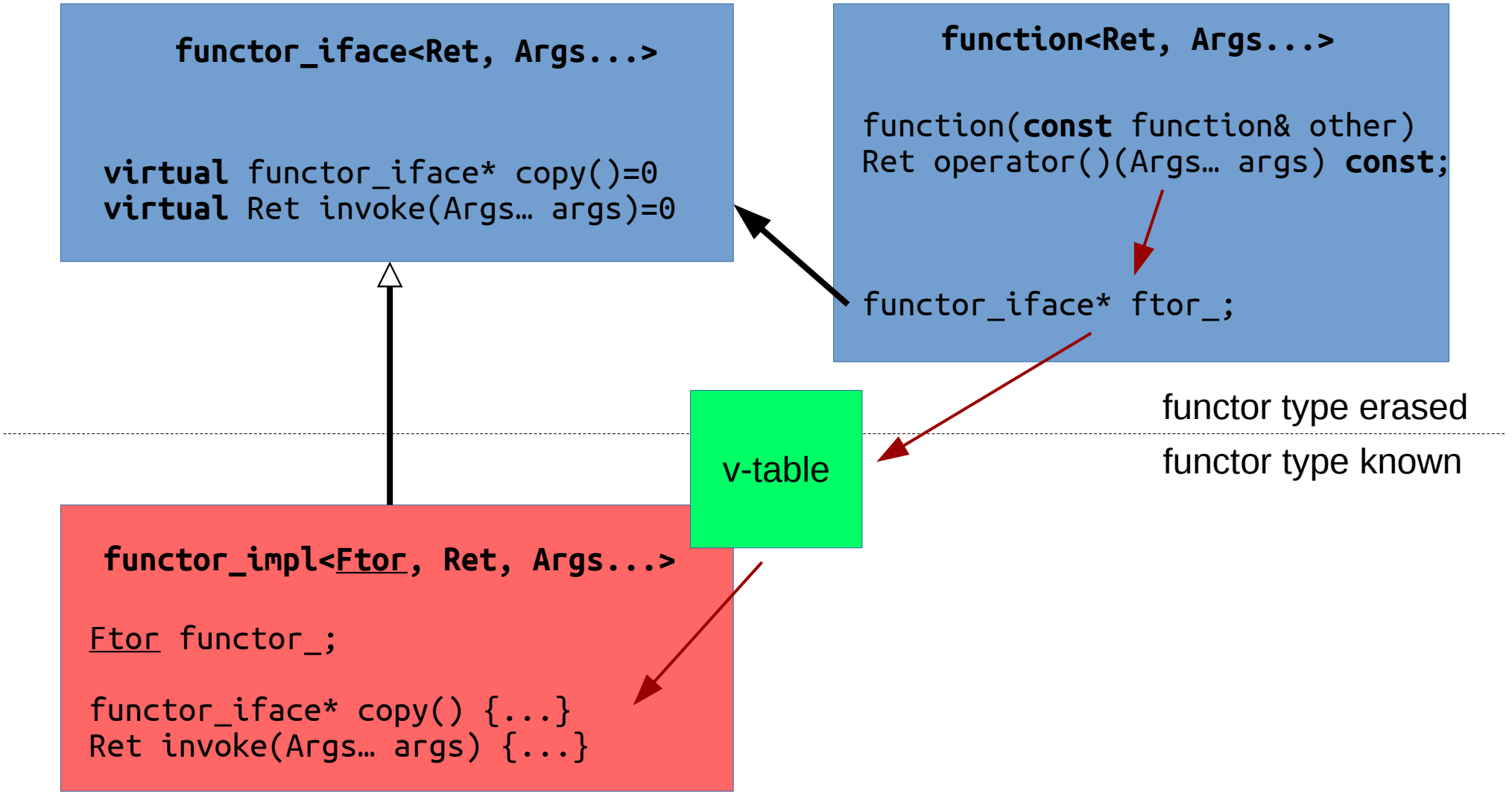
Type erasure - implementation

```
functor_impl* functor_impl::copy() const
{
   return new functor_impl(*this);
}

Ret functor_impl::invoke(Args... &&args)
{
   return functor(std::forward<Args>(args)...);
}
```

Type erasure



Type erasure



Writing our own

```
template<typename Ret, typename Args...>
class function<Ret(Args...)>
{
public:
   // ...

private:
   std::unique_ptr<functor_iface> ftor_;
};
```

Writing our own

```
// Initialization constructor
template<typename Ftor>
void function(Ftor f) {
    ftor_ = std::make_unique<functor_impl<Ftor>>(f);
}
```

Writing our own

```
// Invocation operator
Ret operator()(Args... args) const {
   if (!ftor_)
      throw std::bad_function_call();
   ftor_->invoke(std::forward<Args>(args)...);
}
```

Writing our own

```
// convert-to-bool operator
explicit operator bool() const {
    return !!ftor_;
}

// DONE! (mostly)
```

Performance

Benchmark result

```
Run on (4 X 3392.3 MHz CPU s)
2016-10-17 20:38:42
Benchmark                                      Time           CPU Iterations
----------------------------------------------------------------------------
std_function/std_function_holding_functor    190 ns        190 ns    3643877
std_function/std_function_holding_function   244 ns        244 ns    2863908
pr_function/pr_function_holding_functor      196 ns        196 ns    3648524
pr_function/pr_function_holding_function     242 ns        242 ns    2831490
raw_functor                                   48 ns         48 ns   14739537
raw_function                                 164 ns        164 ns    4242726
virtual_fun                                  164 ns        164 ns    4247793
```

Counter call, 100x in a loop

Benchmark result

```
Run on (4 X 3392.3 MHz CPU s)
2016-10-17 20:38:42
Benchmark                                          Time           Comment
-------------------------------------------------------------------------------
std_function/std_function_holding_functor        190 ns  not-inlined + if
std_function/std_function_holding_function       244 ns  not-inlined + if + ptr
pr_function/pr_function_holding_functor          196 ns  not-inlined + if
pr_function/pr_function_holding_function         242 ns  not-inlined + if + ptr
raw_functor                                       48 ns  inlined
raw_function                                     164 ns  not-inlined
virtual_fun                                      164 ns  not-inlined
```

Counter call, 100x in a loop

Conclusions

- Nothing magical inside
- Just a library container
- Not unreasonably slow

Questions ?

maciej.gajewski0@gmail.com
https://github.com/maciekgajewski/demystifying_std_function