C++ Type Erasure Demystified

Fedor G Pikus
Siemens Fellow
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
What is type erasure?

- Type erasure is a programming technique by which the explicit type information is removed from the program. It is a type of abstraction that ensures that the program does not explicitly depend on some of the data types.
What is type erasure?

- Type erasure is magic.
What is type erasure?

- Type erasure is a programming technique by which the explicit type information is removed from the program. It is a type of abstraction that ensures that the program does not explicitly depend on some of the data types.
- A program is written in a strongly typed language but does not use the actual types. How?
- Why, by abstracting away the type, of course!
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
How does type erasure look like?

- How does a program with explicit types look like?

```cpp
std::unique_ptr<int> p(new int(0));
```

- Creates and deletes an integer
- Deletion is not explicitly visible
  - Done by `std::default_delete` which calls `operator delete`
  - How does a program with `explicit` types look like?
How does type erasure look like?

• How does a program with explicit types look like?

```cpp
class MyHeap {
    ...
    void* allocate(size_t size);
    void deallocate(void* p);
};

void* operator new(size_t size, MyHeap* heap) {
    return heap->allocate(size);
}
```

• Custom operator new(size_t, MyHeap*) for MyHeap
  - no operator delete with arguments
How does type erasure look like?

- How does a program with explicit types look like?

```cpp
struct MyDeleter {
    ...
    template <typename T> void operator()(T* p);
};
{
    MyHeap heap;
    std::unique_ptr<int, MyDeleter> p(new(&heap) int(0),
        MyDeleter(&heap));
}
```

- Creates and deletes an integer, allocation from heap
How does type erasure look like?

- How does a program with explicit types look like?

```cpp
class MyDeleter {
    MyHeap* heap_;
public:
    MyDeleter(MyHeap* heap) : heap_(heap) {}
    template <typename T> void operator()(T* p) {
        p->~T();
        heap_->deallocate(p);
    }
};
```

- No-throw movable (or copyable)
Show me the explicit types

- Types are explicitly present in the program

MyHeap heap;
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), MyDeleter(&heap));
std::unique_ptr<int> q(new int(0));
p = std::move(q); // Error: p and q are different types
std::unique_ptr<int> p(new(&heap) int(0), // default_delete is not MyDeleter
                      MyDeleter(&heap)); //

- Unique pointers to different types are different types – of course
- Unique pointers to the same type but with different deleters are different types too (we can deduce deleter type from the pointer type)
Where is type erasure already?

- How is shared pointer different from unique pointer?

```cpp
std::unique_ptr<int> p(new int(0));
std::shared_ptr<int> q(new int(0));
```

- Now with custom deleter:

```cpp
MyHeap heap;
std::unique_ptr<int, MyDeleter> p(new(&heap) int(0), MyDeleter(&heap));
std::shared_ptr<int> q(new(&heap) int(0), MyDeleter(&heap));
```

- Where is the deleter type? Erased!

- We cannot deduce deleter type from the pointer type
Is the erased type gone?

- Shared pointers with different deleters have the same type

```
std::shared_ptr<int> p(new int(0));
MyHeap heap;
std::shared_ptr<int> q(new(&heap) int(0), MyDeleter(&heap));
q = p;  // OK, same type
```

- But each shared pointer invokes the correct deleter

- Erased types are not explicitly visible in the program (no decltype in your code depends on the erased type)

- Actions that depend on these types are performed correctly
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
General type erasure

std::function<size_t(const std::string&)> f;

- Any function that can be called with a string and returns a size_t

size_t f1(const std::string& s) { return s.capacity(); }
f = f1;
f = [=](const std::string& s) { return s.find(c); };
f = &std::string::size;

- f1, lambda, and member function all have different types
  - f has only one type but can store any of these callable objects

- Type erasure is an abstraction for multiple implementations that provide the same behavior (the relevant behavior is what matters)
Type erasure as a design pattern

- What problem does type erasure solve
  - The code expects certain behavior
  - The code is written in terms of an abstraction that provides this behavior
  - Many concrete types can implement this behavior
  - All properties of these types that are not relevant to the behavior are erased
    - Starting with the name of the type

- Type erasure separates the interface from the implementation
  - So does inheritance, but type erasure does not require common base class
  - Type erasure is non-intrusive
  - External polymorphism (types do not have to be designed for it)
Outline

• What is type erasure
• What does type erasure look like
• Type erasure as a design pattern
• Type erasure as an implementation technique
• How does it work?
• Three ways to implement type erasure in C++
  – Inheritance
  – Static functions
  – V-table
• Performance benchmarks
Type erasure for decoupling dependencies

- Type erasure can be an implementation technique

```cpp
class Network {
    void send(const char* data);
    void receive(const char* buffer);
};
```

- Network is used by our entire application
  - but one small part needs encryption
  - and another small part has bad network and must use error correction
Type erasure for decoupling dependencies

- Packets may need additional processing

```cpp
class Network {
    bool needs_processing_; 
    void send(const char* data) {
        if (needs_processing_) apply_processing(data);
    }
};
```

- Network is used by our entire application
- All of which now depends on the processing code
- Processors may be of different types
Type erasure for decoupling dependencies

- Type erasure offers a simple solution

```cpp
class Network {
    std::function<const char*(const char*)> processor = 
        [](const char* c){ return c; };
    void send(const char* data) {
        data = processor(data);
    }
    template <typename F> void SetProcessor(F&& f) {
        processor = std::forward<F>(f);
    }
};
```

- Does not depend on processor type!
- One and only mention of processor type
- Processor type erased
Type erasure for decoupling dependencies

- Type erasure here implements Strategy pattern
  - Implementation of a particular behavior can be chosen at run time

Network N;
N.SetProcessor(
    [](const char* s){
        char* c;
        ... process the input ...;
        return c;
    }
);

Network N;
N.SetProcessor(
    [](const char* s){
        char* c;
        ... process the input ...;
        return c;
    }
);
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
How does the type erasure work?

- You already know how:
  ```c
  void qsort(void *base, size_t nmemb, size_t size,
              int (*compare)(const void *, const void *));
  ```
  ```c
  int less(const void* a, const void* b) {
    return *(const int*)a - *(const int*)b;
  }
  ```
  ```c
  int a[10] = { 1, 10, 2, 9, 3, 8, 4, 7, 5, 0 };
  qsort(a, 10, sizeof(int), less);
  ```

- The code depends only on the relevant type properties:
  - size and how to compare types
How does the type erasure work?

- Type erasure in C:

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void *, const void *));
```

- The general code does not depend on the type to sort
- The code depends only on the relevant type properties:
  - size and how to compare types
- All interfaces are generic – no type information
How does the type erasure work?

- Type erasure in C:

```c
int a[10] = { 1, 10, 2, 9, 3, 8, 4, 7, 5, 0 };
qsort(a, 10, sizeof(int), less);
```

- At the call site, the specific types are known
- They may be used to compute some properties (often size)
- All other type information is erased
  - From this point forward, we execute the code that has no knowledge of the specific type it was called with
How does the type erasure work?

- Type erasure in C:
  ```c
  int less(const void* a, const void* b) {
    return *(const int*)a - *(const int*)b;
  }
  ```

- Type must be recovered at some point
  - where the type-specific actions take place

- Type reification (recovery) is manual in C
  - No compile-time or run-time error detection

- C++ helps with [only] that!
  - C++ automates type reification and makes it correct by construction
The mechanism of type erasure

- The general code does not depend on the erased type
  - Type properties like size are sometimes used
- The call site is the last place where the actual type is known
- Type is reified when the type-specific action must be performed
- Type is hidden in the code of the function that performs this action
- The function is invoked through a type-agnostic interface
  - The type-dependent code converts from abstract to concrete type
- In C, the type-dependent code is written manually
- In C++, we can make the compiler generate the correct code
  - That’s “all” C++ adds to type erasure
Type erase implementations

- Three main implementations
  - Using inheritance
  - Using static functions
  - Using v_table
- All done using the shared pointer as the example
- Focus on the deleter, not the shared ownership
- Same as type-erased unique pointer for our purposes
- Owning type-erased smart pointer: smartptr
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
template <typename T> class smartptr_te {
public:
    template <typename Deleter> smartptr_te(T* p, Deleter d) :
        p_(p), ??? {} 
~smartptr_te() { ??? delete p using d ??? }
T* operator->() { return p_; }
const T* operator->() const { return p_; }
private:
    T* p_; 
??? something about deleter ???
};
Type erasure implementation #1

```cpp
template <typename T> class smartptr_te {
    // Only one template parameter – no deleter in the type of smartptr
    // - No way to deduce deleter type from smartptr_te type
```
Type erasure implementation #1

template <typename T> class smartptr_te {

• Only one template parameter – no deleter in the type of smartptr

public:

template <typename Deleter> smartptr_te(T* p, Deleter d) :
    p_(p), ??? {} 

• This constructor is the last place where the type Deleter is known
  – Deleter is not a part of the smartptr_te type

• From this point on, the type is erased and the code is generic

• The constructor must generate some Deleter-specific code
  – And hook it up to to the generic call in the destructor
Type erasure implementation #1 - Inheritance

- Erased type Deleter is hidden in a polymorphic derived class:

```cpp
struct destroy_base {
    virtual void operator()(void*) = 0;
    virtual ~deleter_base() {}
};

template <typename Deleter>
struct destroy : public destroy_base {
    destroy (Deleter d) : d_(d) {}
    void operator()(void* p) override { d_(static_cast<T*>(p)); }
    Deleter d_;  // must use void* to match base
};
```

abstract base (void* only)
derived class template
must use void* to match base
Type erasure implementation #1 - Inheritance

- Erased type Deleter is hidden in a polymorphic derived class

```cpp
template <typename T> class smartptr_te {
    struct destroy_base { ... };
    template <typename Deleter>
    struct destroy : public destroy_base { ... };
    public:
    template <typename Deleter> smartptr_te(T* p, Deleter d) :
        p_(p), d_(new destroy<Deleter>(d)) {}
    ~smartptr_te() { (*d_)(p_); delete d_; }
    destroy_base* d_;
};
```
Type erasure implementation #1 - Inheritance

- The constructor of the smart pointer is where we know the type:
  template <typename Deleter> smartptr_te(T* p, Deleter d)
- In the constructor, we create the code that knows the right type:
  destroy_base d_ = new destroy<Deleter>(d)
- The reifying code has type-agnostic interface:
  template <typename Deleter> struct destroy ...
  {
    void operator()(void* p) override { d_(static_cast<T*>(p)); } 
  };
- This type-agnostic interface is called from the generic code:
  ~smartptr_te() { (*d_)(p_); }
Type erasure implementation #1 - Inheritance

- What about default deleter?
  - Bad: leave deleter pointer null, check, and call std::default_delete
  - Good: default-initialize the deleter

```cpp
smartptr_te(T* p) : p_(p) {}  
destroy_base d_ = new destroy<std::default_delete<T>>(
    std::default_delete<T>{});
```

- The rest of the code keeps the same logic!
Type erasure implementation #1 - Inheritance

- Type-erased class does not depend on the erased type
  - `std::shared_ptr<T>, std::function<F>, std::any`
- The constructor is a template and deduces the type to be erased
  - For the smart pointer, it’s the deleter
- The constructor creates a derived object with the override such that:
  - The body of the function uses erased type and is correct by construction
  - The interface of the function is type-agnostic
- The derived object is accessed through the base pointer
- If a default value for the erased type is allowed, the base pointer is default-initialized with the default action
Type erasure implementation #1 - Inheritance

• How to do other common operations on type-erased objects?
• Copying: the destroy hierarchy needs a virtual clone() function
• Moving: transfer the deleter to the new object
• Comparison:
  – For smart pointers, often only addresses are compared
  – In general, need another virtual function to compare deleters
• In general, if we need to support an operation on type-erased objects that is affected by the erased type, we have to add a virtual function to the base class and specific overrides to the derived class template
  – Each of these virtual functions needs to reify the type
Type erasure is slow!

- Our implementation of type erasure has a glaring inefficiency: memory is allocated when the type is erased (`new destroy<Deleter>`)!
- The common solution is the local buffer optimization:

```cpp
template <typename T> class smartptr_te {
    template <typename Deleter> smartptr_te(T* p, Deleter d) :
        p_(p), d_(new(buf_) destroy<Deleter>(d)) {}
    ~smartptr_te() { (*d_)(p_); d_->~destroy_base(); }
    alignas(8) char buf_[16];
    destroy_base* d_;}
```
Type erasure is fast but [may be] broken

- Local buffer size does not depend on the erased type Deleter
  - The whole point of type erasure is that `smartptr_te` does not depend on it
- When the constructor is called, Deleter may or may not fit into `buf_`
- What happens if the erased type does not fit into the local buffer?
  1) The implementation switches to dynamic memory allocations
  2) `static_assert` in the compiler
- `std::function` uses local buffer and option 1
- Many high-performance implementations use option 2
  - Make the buffer size a template parameter of the class
Type erasure with local buffers

- Local buffer optimization is often used with type erasure
- Avoids dynamic allocations if the type fits into the buffer
- Incurs slight overhead when dynamic allocation is still done
- Design decision: allow all types or only small enough types?
  - Enforced local buffer makes “slow path” a compile-time error
- “Moving” objects with local buffers often becomes copying
  - Size is small, so copy is cheap
  - Copy operations may throw when move is noexcept
- Design decision: restrict the optimization to noexcept-copyable types?
Type erasure using inheritance

- At the point where the type is erased, the compiler instantiates a class template that depends on the erased type
  - Often the constructor of the type-erased class
- All template instantiations inherit from the common base
  - The polymorphic interface is type-agnostic (void*)
- Template generates correct-by-construction member functions that reify the erased type, usually through casts
  - Erased type is hidden in the generated code
- Both primary (deleter) and secondary (copy, move, compare) operations are implemented through virtual overrides
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
Type erasure implementation #2

- This technique is much closer to the C type erasure:

```cpp
void reify_func(void* p) {
    TE* q = static_cast<TE*>(p); ... do work on type TE ...
}
```

- C++ helps to generate correct-by-construction function:

```cpp
template <typename TE>
void reify_func(void* p) {
    TE* q = static_cast<TE*>(p); ... do work on type TE ...
}
```

- The signature is type-agnostic – any instantiation can be assigned to

```cpp
void(*)(void*) fp = reify_func<MyDeleter>;
```
template <typename T> class smartptr_te_static {
    void(*)(T*, void*) destroy_;  
    template <typename Deleter>     
    static void invoke_destroy(T* p, void* d) {  
        (*static_cast<Deleter*>(d))(p);  
    }
public:
    template <typename Deleter>     
    smartptr_te_static(T* p, Deleter d)  
    : p_(p), destroy_(invoke_destroy<Deleter>()) { ... } 
};
template <typename T> class smartptr_te_static {
    alignas(8) char buf_[8];
public:
    template <typename Deleter>
    smartptr_te_static(T* p, Deleter d)
        : p_(p), destroy_(invoke_destroy<Deleter>) {
            ::new (static_cast<void*>(buf_)) Deleter(d);
        }
    ~smartptr_te_static() { this->destroy_(p_, buf_); }
};
template <typename T> class smartptr_te_static {
    public:
    template <typename Deleter>
    smartptr_te_static(T* p, Deleter d)
        : p_(p), destroy_(invoke_destroy<Deleter>) {
            ::new (static_cast<void*>(buf_)) Deleter(d);
            static_assert(sizeof(Deleter) <= sizeof(buf_));
            ... also trivially destructible, copyable, etc.
        }
};
Type erasure implementation #2

- Static functions generated by the template are “code only”
- Only the deleter needs to be stored (not a composite object)
- Local buffer optimization avoids memory allocation costs (same as before)
- Dynamically allocated buffers can be used for larger types
- The downside: how do we copy or destroy the deleter?
  1) Limit to trivially destructible/copyable types (often OK in practice)
  2) Add another static function to destroy the type-erased deleter
     • And another one for copying…
- This implementation is fast but gets bloated if many operations are abstracted via type erasure
Type erasure using static functions

- At the point where the type is erased, the compiler instantiates a static function template that depends on the erased type
  - Erased type is hidden in the generated code
- All template instantiations have the same signature
  - The signature of the function is type-agnostic (void*)
- Template instantiation is assigned to a function pointer
- Objects with state are stored in local or dynamic memory
- Type-erased code is executed by an indirect function call
- For each supported operation, a function pointer is needed
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
Type erasure implementation #3

- This implementation is similar to option 2 (static function) but supports multiple operations without a function pointer for each one.

```cpp
template<typename T> class smartptr_te_vtable {
    struct vtable_t {
        using destroy_t = void(*)(T*, void*);
        using destructor_t = void(*)(void*);
        destroy_t destroy_;  
        destructor_t destructor_; 
    };
    const vtable_t* vtable_ = nullptr;
    ...
}
```

- type-agnostic signatures
- function pointers
- incoming: memory allocation?
template<typename T> class smartptr_te_vtable {
    const vtable_t* vtable_ = nullptr;
    template<typename Deleter>
    constexpr static vtable_t vtable = { ... };

    public:
    template<typename Deleter>
    smartptr_te_vtable(T* p, Deleter d)
        : p_(p), vtable_(&vtable<Deleter>()) {
    }
};
Type erasure implementation #3

template <typename T> class smartptr_te_vtable {
    const vtable_t* vtable_ = nullptr;
    template <typename Deleter>
    constexpr static vtable_t vtable = { ... };
    ...
    vtable_(&vtable<Deleter>)
};

• Instantiating vtable on each Deleter type creates a static variable
• Template static variables do not need definitions in .C files
• Class smartptr_te_vtable<T> has many static variables all named vtable
template <typename T> class smartptr_te_vtable {
    template <typename Deleter>
    constexpr static vtable_t vtable = {
        smartptr_te_vtable::template destroy<Deleter>,
        smartptr_te_vtable::template destructor<Deleter>
    };
    template <typename Deleter>
    static void destroy(T* p, void* d) {
        (*static_cast<Deleter*>(d))(p);
    }
};
Type erasure implementation #3

template <typename T> class smartptr_te_vtable {
    template <typename Deleter>
    constexpr static vtable_t vtable = {
        smartptr_te_vtable::template destroy<Deleter>,
        smartptr_te_vtable::template destructor<Deleter>
    };

    template <typename Deleter>
    static void destructor(void* d) {
        static_cast<Deleter*>(d)->~Deleter();
    }
};
template <typename T> class smartptr_te_vtable {
    const vtable_t* vtable_ = nullptr;
    alignas(8) char buf_[8];

public:
    template <typename Deleter>
    smartptr_te_vtable(T* p, Deleter d)
        : p_(p), vtable_(&vtable<Deleter>) {
        static_assert(sizeof(Deleter) <= sizeof(buf_));
        ::new (static_cast<void*>(buf_)) Deleter(d);
    }
};
Type erasure implementation #3

```cpp
template <typename Deleter>
smartptr_te_vtable(T* p, Deleter d)
 : p_(p), vtable_(&vtable<Deleter>) {
  static_assert(sizeof(Deleter) <= sizeof(buf_));
  ::new (static_cast<void*>(buf_)) Deleter(d);
}
```

- Constructor does three things:
  - Store the object pointer \( p \)
  - Store the deleter in the buffer
  - Point vtable to the right static variable
template <typename T> class smartptr_te_vtable {
    const vtable_t* vtable_ = nullptr;
    alignas(8) char buf_[8];
public:
    ~smartptr_te_vtable() {
        this->vtable_->destroy_(p_, buf_);
        this->vtable_->destructor_(buf_);
    }
};

struct vtable_t {
    using destroy_t = void(*)(T*, void*);
    using destructor_t = void(*)(void*);
    destroy_t destroy_;
    destructor_t destructor_;}

• Copy etc are handled similarly
• Only one vtable pointer in the class!
Type erasure using v-table

- At the point where the type is erased, the compiler generates multiple correct-by-construction reification functions
  - The erased type is hidden in the code of these functions
- The signature of all functions is type-agnostic (void*)
- All function pointers are stored in a static variable
  - Template static variable, depends on the deleter, constructor instantiates it
- The vtable pointer is set to the right static vtable variable
- The deleter is saved in a buffer (local or dynamic)
- Type Deleter has been erased: we have f(void*) and char[]
- All reification functions are invoked through their pointers in the vtable
- This really is how compilers build v-tables!
Type erasure using v-tables

- At the point where the type is erased, the compiler instantiates a static variable that depends on the erased type.
- Initializing this variable instantiates function template on the same type.
  - Erased type is hidden in the generated code.
- All function template instantiations have the same signature.
- All static variable instantiations have the same type.
- Objects with state are stored in local or dynamic memory.
- Type-erased code is executed by an (double) indirect function call.
- For each supported operation, a function pointer in the vtable is needed.
- There is only one vtable pointer in the object.
Outline

- What is type erasure
- What does type erasure look like
- Type erasure as a design pattern
- Type erasure as an implementation technique
- How does it work?
- Three ways to implement type erasure in C++
  - Inheritance
  - Static functions
  - V-table
- Performance benchmarks
## Performance of type erasure

### Smart pointer creation and deletion

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>CPU</th>
<th>Iterations</th>
<th>UserCounters...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_rawptr</td>
<td>8.88 ns</td>
<td>8.88 ns</td>
<td>81125063 items_per_second=112.561M/s</td>
<td></td>
</tr>
<tr>
<td>BM_uniqueptr</td>
<td>9.12 ns</td>
<td>9.12 ns</td>
<td>76533789 items_per_second=109.678M/s</td>
<td></td>
</tr>
<tr>
<td>BM_sharedptr</td>
<td>21.5 ns</td>
<td>21.5 ns</td>
<td>32544564 items_per_second=46.5102M/s</td>
<td></td>
</tr>
<tr>
<td>BM_make_sharedptr</td>
<td>12.0 ns</td>
<td>12.0 ns</td>
<td>58151720 items_per_second=83.2405M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr</td>
<td>9.37 ns</td>
<td>9.34 ns</td>
<td>76504653 items_per_second=107.093M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr_te</td>
<td>19.7 ns</td>
<td>19.7 ns</td>
<td>35678838 items_per_second=50.8338M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr_te_lb_opt</td>
<td>10.6 ns</td>
<td>10.6 ns</td>
<td>65890276 items_per_second=94.0315M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr_te_lb_only</td>
<td>10.5 ns</td>
<td>10.5 ns</td>
<td>65952774 items_per_second=94.8626M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr_te_static</td>
<td>9.86 ns</td>
<td>9.86 ns</td>
<td>70883148 items_per_second=101.38M/s</td>
<td></td>
</tr>
<tr>
<td>BM_smartptr_te_vtable</td>
<td>10.6 ns</td>
<td>10.6 ns</td>
<td>60152866 items_per_second=94.6588M/s</td>
<td></td>
</tr>
</tbody>
</table>
Performance of type erasure

- Deleter performance of smart pointers is not all that exciting
- What’s the most performance-critical type-erased object?
- std::function
  - Type erasure machinery is exercised on every call!
- How to implement a type-erased function?
Performance of type erasure

- Deleter performance of smart pointers is not all that exciting
- What’s the most performance-critical type-erased object?
- std::function
  - Type erasure machinery is exercised on every call!
- How to implement a type-erased function?
Performance of type erasure

- How to implement a type-erased function?
- Use the fastest option 2 (static function) for the call
- Use option 3 (vtable) for copy, move, delete, etc (no object to store)
- Use local buffer optimization (only? - design decision)
- Optimize for trivially-everything objects? (another design decision)

Implementation based on works by Arthur O'Dwyer and Eduardo Madrid
  - Zoo project (https://github.com/thecppzoo/zoo)
Type-erased nonstd::function

```cpp
template<typename Signature, size_t Size = 16, size_t Alignment = 8> struct Function;
```

- Size and Alignment are for the local buffer (ignore for SlideWare)

```cpp
template<typename Signature> class Function;
```

```cpp
template<typename Res, typename... Args>
struct Function<Res(Args...)> {...};
```

- Partial specialization is convenient to extract return type and parameters
  - Reflection, yay..!

incomplete type
Type-erased nonstd::function

- Local buffer (remember Size and Alignment?)

```cpp
template<typename Res, typename... Args>
struct Function<Res(Args...)> {
    alignas(Alignment) char space_[Size];
    ...
};
```

- This is where we store the callable object
  - std::function has an 8-byte buffer
  - enough for function pointers and stateless callables like lambdas
  - function pointers require 16 bytes
Type-erased nonstd::function

- Function call uses the static function method:

```
using executor_t = Res(*)(Args..., Function*);
executor_t executor_;
```

```
template<typename Callable>
static Res executor(Args... args, Function* this_function)
{
    return (*reinterpret_cast<Callable*>(
        this_function->space_))
        (std::forward<Args>(args)...);
}
```

- Type-agnostic signature
- Reification template function
- Function pointer
- Restore Callable
- Invoke Callable with args

70 Type Erasure
Type-erased nonstd::function

- Constructor hides erased type in the code it generates:

  ```cpp
  template <typename CallableArg,
             typename Callable = std::decay_t<CallableArg>>
  requires(!std::same_as<Function, Callable>)
  Function(CallableArg&& callable) : executor_(executor<Callable>)
  {
    ::new (static_cast<void*>(space_))
    Callable(std::forward<CallableArg>(callable));
  }
  ```

- Store the callable in the buffer (strip references)
- Generate reification function and set the function pointer (`executor_`) not a copy ctor
Type-erased nonstd::function

- The call operator invokes the executor with the specified arguments
- The arguments do not have to match the function signature but must be convertible to those
  - use concepts or static asserts for better error messages

```cpp
template <typename... CallArgs>
Res operator()(CallArgs&&... callargs) const {
    return this->executor_(std::forward<CallArgs>(callargs)...,
                           const_cast<Function*>(this));
}
```

Template instantiation created during construction
Type-erased \texttt{nonstd::function}

- How to deal with copy, move, and destruction?
  1) Implement using vtable
  2) Restrict to trivially-everything types

```cpp
template <typename CallableArg,
          typename Callable = std::decay_t<CallableArg>>
Function(CallableArg&& callable) : executor_(executor<Callable>) {
  static_assert(sizeof(Callable) <= Size);
  static_assert(alignof(Callable) <= Alignment);
  static_assert(std::is_trivially_destructible<Callable>::value);
  static_assert(std::is_trivially_copyable<Callable>::value);
  ...
}
```

73  
Type Erasure
Type-erased nonstd::function

- std::function can be default-constructed (nothing to call)
  - Throws std::bad_function_call if called anyway
- Bad: default executor_ to null and check at run-time
  - Check is done for all functions, initialized or not
- Good: executor_ is never null, default executor throws

```cpp
static constexpr Res default_executor(Args..., Function*) {
    throw std::bad_function_call();
}
constexpr static executor_t default_executor_ = default_executor;
executor_t executor_ = default_executor_;```

74 Type Erasure
Type-erased nonstd::function

- Destruction, copying, etc are handled by the vtable
- Member functions can be trivially supported
  - needs a constructor overload and another executor template
- Dynamic buffers for large callables are straightforward
- None of these affect performance of the function call
  - Local buffer might, so compare fairly (std::function also uses buffer)
Type erasure using static functions (again)

- At the point where the type is erased, the compiler instantiates a static function template that depends on the erased type
  - Erased type is hidden in the generated code
- All template instantiations have the same signature
- Template instantiation is assigned to a function pointer
- Default function pointer assignment performs the default action
- Objects with state are stored in local or dynamic memory
- Type-erased code is executed by an indirect function call
- Other, less performance-critical operations are handled using vtable
Type-erased nonstd::function performance

- Let’s see what a call to a function looks like:

```cpp
int f(int a, int b, int c, int d);
using F = int(int, int, int, int);
auto F_invoke(int a, int b, int c, int d, F f) {
    return f(a, b, c, d);
}
```

- Assembly of `F_invoke`:

```
0000000000000000 <_Z8F_invokeiiiiPFiiiiiE>:
  0:  41 ff e0                jmpq   *%r8
```
Type-erased nonstd::function performance

- Now let’s see what a call to a std::function looks like:

```cpp
int f(int a, int b, int c, int d);
using F = int(int, int, int, int);
using SF = std::function<F>;
auto SF_invoke(int a, int b, int c, int d, const SF& f) {
    return f(a, b, c, d);
}
```

- Assembly of SF_invoke:

```
0000000000000000 <_Z9SF_invoke>:
   0:   48 83 ec 18    sub    $0x18,%rsp
   4:   49 83 78 10 00 cmpq   $0x0,0x10(%r8)
   9:   89 3c 24      mov    %edi,(%rsp)
   c:   89 74 24 04   mov    %esi,0x4(%rsp)
   10:  89 54 24 08   mov    %edx,0x8(%rsp)
   14:  89 4c 24 0c   mov    %ecx,0xc(%rsp)
   18:  74 20        je     3a <_Z9SF_invoke>
   1a:  4c 89 c0      mov    %r8,%rax
   1d:  48 8d 4c 24 08 lea    0x8(%rsp),%rcx
   22:  4c 8d 44 24 0c lea    0xc(%rsp),%r8
   27:  48 89 e6      mov    %rsp,%rsi
   2a:  48 8d 54 24 04 lea    0x4(%rsp),%rdx
   2f:  48 89 c7      mov    %rax,%rdi
   32:  ff 50 18      callq  *0x18(%rax)
   35:  48 83 c4 18   add    $0x18,%rsp
   39:  c3           retq
   3a:  e8 00 00 00 00 callq  3f <_Z9SF_invoke>
```
Type-erased nonstd::function performance

- Assembly of SF_invoke:

```
0000000000000000 <__Z9SF_invoke>:

0:  48 83 ec 18  sub    $0x18,%rsp
4:  49 83 78 10 00 cmpq $0x0,0x10(%r8)
9:  89 3c 24  mov    %edi,(%rsp)
c:  89 74 24 04 mov    %esi,0x4(%rsp)
10: 89 54 24 08 mov    %edx,0x8(%rsp)
14: 89 4c 24 0c mov    %ecx,0xc(%rsp)
18: 74 20  je 3a <__Z9SF_invoke>
1a: 4c 89 c0 mov    %r8,%rax
1d: 48 8d 4c 24 08 lea 0x8(%rsp),%rcx
22: 4c 8d 44 24 0c lea 0xc(%rsp),%r8
27: 48 89 e6 mov  %rsp,%rsi
2a: 48 8d 54 24 04 lea 0x4(%rsp),%rdx
2f: 48 89 c7 mov  %rax,%rdi
32: ff 50 18 callq *0x18(%rax)
35: 48 83 c4 18 add  $0x18,%rsp
39: c3 retq
3a: e8 00 00 00 00 callq 3f <__Z9SF_invoke>
```
Type-erased nonstd::function performance

- OK, so what does a call to our type-erased function looks like?
  
  ```cpp
  int f(int a, int b, int c, int d);
  using F = int(int, int, int, int);
  using FF = Function<F>;
  auto FF_invoke(int a, int b, int c, int d, const FF& f) {
    return f(a, b, c, d);
  }
  ```

- Assembly of `FF_invoke`:
  
  ```assembly
  0000000000000000 <_Z9FF_invokeiiiiRK8FunctionEE>:  
  0: 41 ff 60 10 jmpq  *0x10(%r8)
  ```
Type-erased nonstd::function real performance

- Better assembly does not always translate into better performance
- We must benchmark the call itself

---

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>CPU</th>
<th>Iterations</th>
<th>UserCounters...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_F_invoke</td>
<td>25.3 ns</td>
<td>25.3 ns</td>
<td>27518802 items_per_second=1.26442G/s</td>
<td></td>
</tr>
<tr>
<td>BM_FF_invoke</td>
<td>26.0 ns</td>
<td>26.0 ns</td>
<td>26985686 items_per_second=1.22865G/s</td>
<td></td>
</tr>
<tr>
<td>BM_SF_invoke</td>
<td>53.7 ns</td>
<td>53.7 ns</td>
<td>12798869 items_per_second=596.021M/s</td>
<td></td>
</tr>
</tbody>
</table>
### Type-erased nonstd::function real performance

- How does it compare with a regular or virtual function call?
  - The function body is in a separate compilation unit in all cases

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>CPU</th>
<th>Iterations</th>
<th>UserCounters...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_F_invoke</td>
<td>25.3 ns</td>
<td>25.3 ns</td>
<td>27518802</td>
<td>items_per_second=1.26442G/s</td>
</tr>
<tr>
<td>BM_FF_invoke</td>
<td>26.0 ns</td>
<td>26.0 ns</td>
<td>26985686</td>
<td>items_per_second=1.22865G/s</td>
</tr>
<tr>
<td>BM_SF_invoke</td>
<td>53.7 ns</td>
<td>53.7 ns</td>
<td>12798869</td>
<td>items_per_second=596.021M/s</td>
</tr>
<tr>
<td>BM_f</td>
<td>26.0 ns</td>
<td>26.0 ns</td>
<td>27341020</td>
<td>items_per_second=1.23192G/s</td>
</tr>
<tr>
<td>BM_virtual</td>
<td>25.9 ns</td>
<td>25.9 ns</td>
<td>27123930</td>
<td>items_per_second=1.236G/s</td>
</tr>
</tbody>
</table>
Type-erased `nonstd::function` real performance

- The cost of a (well-done) indirection is about the same
- Nothing beats the performance boost from inlining

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>CPU</th>
<th>Iterations</th>
<th>UserCounters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_F_invoke</td>
<td>25.3 ns</td>
<td>25.3 ns</td>
<td>27518802 items_per_second=1.26442G/s</td>
<td></td>
</tr>
<tr>
<td>BM_FF_invoke</td>
<td>26.0 ns</td>
<td>26.0 ns</td>
<td>26985686 items_per_second=1.22865G/s</td>
<td></td>
</tr>
<tr>
<td>BM_SF_invoke</td>
<td>53.7 ns</td>
<td>53.7 ns</td>
<td>12798869 items_per_second=596.021M/s</td>
<td></td>
</tr>
<tr>
<td>BM_f</td>
<td>26.0 ns</td>
<td>26.0 ns</td>
<td>27341020 items_per_second=1.23192G/s</td>
<td></td>
</tr>
<tr>
<td>BM_inline</td>
<td>0.209 ns</td>
<td>0.209 ns</td>
<td>10000000000 items_per_second=153.049G/s</td>
<td></td>
</tr>
<tr>
<td>BM_virtual</td>
<td>25.9 ns</td>
<td>25.9 ns</td>
<td>27123930 items_per_second=1.236G/s</td>
<td></td>
</tr>
</tbody>
</table>
How is type erasure done in C++?

- Type erasure in C++ is very similar to C:
  - The generic code does not have any mention of the erased type
    - Often uses `void*` or `char*`
  - The erased type is hidden in the code of a function that is invoked to perform the type-dependent action
    - The signature of this function is type-agnostic
    - The body of this function reifies the erased type (often with casts)
- C++ automates writing the reification code and ensures that it matches the erased type
- The code with the hidden type is generated by a template at the point where the erased type is last present
What is type erasure?

- Type erasure is used to separate the interface from the implementation
  - Even more: separate relevant interface (type properties) from the rest
  - Other than having the relevant interface, types can be very different
- Type erasure can be used to implement separation of concerns
- Type erasure is often used to break dependencies
- Type erasure doesn’t have to be any more expensive than any other indirection mechanism
  - [with a good implementation] there is no overhead assuming the indirection was needed
- Indirection can be expensive in any guise
- Is decoupling worth the cost of indirection? That is a design decision
C++ Type Erasure Demystified

Questions?
Possibly answers too...