Types Don’t Know

Howard Hinnant
Ripple Labs
CppCon 2014

Overview

Overview

- This talk introduces a new way to hash types.

Overview

- This talk introduces a new way to hash types.
- This technique is being considered for standardization.

rippleLabs

Overview

- This talk introduces a new way to hash types.
- This technique is being considered for standardization.
- Whether or not the committee standardizes this infrastructure, you can implement this in your own applications (free source code available to get you started).

rippleLabs

Scope

rippleLabs

Scope

- This talk will concentrate only on the most important aspects of this new hashing technique.

Scope

- This talk will concentrate only on the most important aspects of this new hashing technique.
- Details will be mentioned in passing, just to let you know the details exist, but will not be covered in depth.

Scope

- This talk will concentrate only on the most important aspects of this new hashing technique.
- Details will be mentioned in passing, just to let you know the details exist, but will not be covered in depth.
- Details are more thoroughly covered in the standard proposal, and are fully fleshed out in the freely-available reference implementation.

Issues

Issues

- How should one “combine” hash codes from your bases and data members to create a “good” hash function?

Issues

- How should one “combine” hash codes from your bases and data members to create a “good” hash function?
- How does one know if you have a good hash function?

Issues

- How should one “combine” hash codes from your bases and data members to create a “good” hash function?
- How does one know if you have a good hash function?
- If somehow you knew you had a bad hash function, how would you change it for a type built out of several bases and/or data members?

rippleLabs

Example Class

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...




};
```

rippleLabs

Example Class

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
```

How does one hash this class?

```
};
```

Example Class
Hash with std::hash<T>

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...




};
```

rippleLabs

Example Class
Hash with std::hash<T>

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
   std::size_t
   hash_code() const
   {
       std::size_t k1 = std::hash<std::string>{}(firstName_);
       std::size_t k2 = std::hash<std::string>{}(lastName_);
       std::size_t k3 = std::hash<int>{}(age_);


};
```

Here is one way…

Example Class
Hash with std::hash<T>

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        std::size_t k1 = std::hash<std::string>{}(firstName_);
        std::size_t k2 = std::hash<std::string>{}(lastName_);
        std::size_t k3 = std::hash<int>{}(age_);
        return hash_combine(k1, k2, k3);  // what algorithm is this?!
    }

};
```

Here is one way…

Is this a **good** hash algorithm?

rippleLabs

Example Class
Hash with std::hash<T>

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
   std::size_t
   hash_code() const
   {
       std::size_t k1 = std::hash<std::string>{}(firstName_);
       std::size_t k2 = std::hash<std::string>{}(lastName_);
       std::size_t k3 = std::hash<int>{}(age_);
       return hash_combine(k1, k2, k3);  // what algorithm is this?!
   }

};
```

What if we wanted to use another hash algorithm?

Is this a **good** hash algorithm?

rippleLabs

Example Hash Algorithm FNV-1A

Another hash algorithm:

```
std::size_t
fnv1a (void const* key, std::size_t len)
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...









};
```

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        std::size_t k1 = fnv1a(firstName_.data(), firstName_.size());
        std::size_t k2 = fnv1a(lastName_.data(), lastName_.size());
        std::size_t k3 = fnv1a(&age_, sizeof(age_));



};
```

rippleLabs

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        std::size_t k1 = fnv1a(firstName_.data(), firstName_.size());
        std::size_t k2 = fnv1a(lastName_.data(), lastName_.size());
        std::size_t k3 = fnv1a(&age_, sizeof(age_));
        return hash_combine(k1, k2, k3);  // what algorithm is this?!
    }
};
```

Ok, but our algorithm is still "polluted" by the combine step…

Anatomy Of A Hash Function

Anatomy Of A Hash Function

1. Initialize internal state.

Anatomy Of A Hash Function

1. Initialize internal state.
2. Consume bytes into internal state.

Anatomy Of A Hash Function

1. Initialize internal state.
2. Consume bytes into internal state.
3. Finalize internal state to result_type (often size_t).

Example Hash Algorithm FNV-1A

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

Example Hash Algorithm FNV-1A

Initialize internal state

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

Example Hash Algorithm FNV-1A

Initialize internal state

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
   std::size_t h = 14695981039346656037u;
   unsigned char const* p = static_cast<unsigned char const*>(key);
   unsigned char const* const e = p + len;
   for (; p < e; ++p)
       h = (h ^ *p) * 1099511628211u;
   return h;
}
```

Consume bytes into internal state

rippleLabs

Example Hash Algorithm FNV-1A

Initialize internal state

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

Consume bytes into internal state

Finalize internal state to size_t

Example Hash Algorithm FNV-1A

Consider repackaging this algorithm to make the three stages separately accessible…

Initialize internal state

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

Consume bytes into internal state

Finalize internal state to size_t

rippleLabs

Example Hash Algorithm FNV-1A

```
std::size_t
fnv1a(void const* key, std::size_t len) noexcept
{
    std::size_t h = 14695981039346656037u;
    unsigned char const* p = static_cast<unsigned char const*>(key);
    unsigned char const* const e = p + len;
    for (; p < e; ++p)
        h = (h ^ *p) * 1099511628211u;
    return h;
}
```

rippleLabs

Example Hash Algorithm FNV-1A

```
class fnv1a
{
    std::size_t h = 14695981039346656037u;
public:
    using result_type = std::size_t;

    void
    operator()(void const* key, std::size_t len) noexcept
    {
        unsigned char const* p = static_cast<unsigned char const*>(key);
        unsigned char const* const e = p + len;
        for (; p < e; ++p)
            h = (h ^ *p) * 1099511628211u;
    }

    explicit
    operator result_type() noexcept
    {
       return h;
    }
};
```

rippleLabs

Example Hash Algorithm FNV-1A

```
class fnv1a
{
    std::size_t h = 14695981039346656037u;
public:
    using result_type = std::size_t;

    void
    operator()(void const* key, std::size_t len) noexcept
    {
        unsigned char const* p = static_cast<unsigned char const*>(key);
        unsigned char const* const e = p + len;
        for (; p < e; ++p)
            h = (h ^ *p) * 1099511628211u;
    }

    explicit
    operator result_type() noexcept
    {
       return h;
    }
};
```

Initialize internal state

Consume bytes into internal state

Finalize internal state to size_t

rippleLabs

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...









};
```

Example Class Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        fnv1a h;




};
```

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        fnv1a h;
        h(firstName_.data(), firstName_.size());
        h(lastName_.data(), lastName_.size());
        h(&age_, sizeof(age_));


};
```

rippleLabs

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        fnv1a h;
        h(firstName_.data(), firstName_.size());
        h(lastName_.data(), lastName_.size());
        h(&age_, sizeof(age_));
        return static_cast<std::size_t>(h); // No more hash_combine!
    }
};
```

Now we are using a “pure” FNV-1A algorithm for the **entire** data structure!

rippleLabs

Example Class
Hash with FNV-1A

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
    std::size_t
    hash_code() const
    {
        fnv1a h;
        h(firstName_.data(), firstName_.size());
        h(lastName_.data(), lastName_.size());
        h(&age_, sizeof(age_));
        return static_cast<std::size_t>(h); // No more hash_combine!
    }
};
```

Now we are using a “pure” FNV-1A algorithm for the **entire** data structure!

This same technique can be used with almost every existing hashing algorithm!

Combining Types

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:




};
```

Combining Types

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:
    std::size_t
    hash_code() const
    {
        std::size_t h1 = customer_.hash_code();
        std::size_t h2 = product_.hash_code();
        std::size_t h3 = date_.hash_code();



};
```

rippleLabs

Combining Types

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:
    std::size_t
    hash_code() const
    {
        std::size_t h1 = customer_.hash_code();
        std::size_t h2 = product_.hash_code();
        std::size_t h3 = date_.hash_code();
        // hash_combine is back!!!
        return hash_combine(h1, h2, h3);
    }
};
```

rippleLabs

Combining Types

How do we use FNV-1A for the **entire** Sale class?

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:
    std::size_t
    hash_code() const
    {
        std::size_t h1 = customer_.hash_code();
        std::size_t h2 = product_.hash_code();
        std::size_t h3 = date_.hash_code();
        // hash_combine is back!!!
        return hash_combine(h1, h2, h3);
    }
};
```

hash_append

Looking back at the Customer class, let's solve this problem!

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
     std::size_t
     hash_code() const
     {
         fnv1a h;
         h(c.firstName_.data(), c.firstName_.size());
         h(c.lastName_.data(), c.lastName_.size());
         h(&c.age_, sizeof(c.age_));
         return static_cast<std::size_t>(h);
     }
};
```

hash_append

Let some other piece of code construct and finalize fnv1a.

Customer only appends to the state of fnv1a.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    // ...
     friend
     void
     hash_append(fnv1a& h, const Customer& c)
     {
         h(c.firstName_.data(), c.firstName_.size());
         h(c.lastName_.data(), c.lastName_.size());
         h(&c.age_, sizeof(c.age_));
     }

};
```

rippleLabs

hash_append

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:









};
```

hash_append

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
public:
    friend
    void
    hash_append(fnv1a& h, const Sale&s)
    {
        hash_append(h, s.customer_);
        hash_append(h, s.product_);
        hash_append(h, s.date_);
    }
};
```

Now types can recursively build upon one another's hash_append to build up state in fnv1a.

hash_append

Primitive and std-defined types can be given hash_append overloads.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:

    friend
    void
    hash_append(fnv1a& h, const Customer& c)
    {
        h(c.firstName_.data(), c.firstName_.size());
        h(c.lastName_.data(), c.lastName_.size());
        h(&c.age_, sizeof(c.age_));
    }
};
```

rippleLabs

hash_append

Primitive and std-defined types can be given hash_append overloads.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:

    friend
    void
    hash_append(fnv1a& h, const Customer& c)
    {
        hash_append(h, c.firstName_);
        hash_append(h, c.lastName_);
        hash_append(h, c.age_);
    }
};
```

Simplify!

hash_append

If all Hash Algorithms follow the same interface as given for fnv1a, then hash_append can be templated on the algorithm.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    template <class HashAlgorithm>
    friend
    void
    hash_append(HashAlgorithm& h, const Customer& c)
    {
        hash_append(h, c.firstName_);
        hash_append(h, c.lastName_);
        hash_append(h, c.age_);
    }
};
```

rippleLabs

hash_append

If all Hash Algorithms follow the same interface as given for fnv1a, then hash_append can be templated on the algorithm.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    template <class HashAlgorithm>
    friend
    void
    hash_append(HashAlgorithm& h, const Customer& c)
    {
        hash_append(h, c.firstName_);
        hash_append(h, c.lastName_);
        hash_append(h, c.age_);
    }
};
```

Now Customer can be hashed using **any** HashAlgorithm!

hash_append

One can easily create a variadic version of hash_append.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    template <class HashAlgorithm>
    friend
    void
    hash_append(HashAlgorithm& h, const Customer& c)
    {
        hash_append(h, c.firstName_);
        hash_append(h, c.lastName_);
        hash_append(h, c.age_);
    }
};
```

hash_append

One can easily create a variadic version of hash_append.

```
class Customer
{
    std::string firstName_;
    std::string lastName_;
    int         age_;
public:
    template <class HashAlgorithm>
    friend
    void
    hash_append(HashAlgorithm& h, const Customer& c)
    {
        hash_append(h, c.firstName_ , c.lastName_ , c.age_);
    }


};
```

hash_append

For primitive types that are *contiguously hashable* one can just send their bytes to the hash algorithm in hash_append.

```
template <class HashAlgorithm>
void
hash_append(HashAlgorithm& h, int i)
{
    h(&i, sizeof(i));
}


template <class HashAlgorithm, class T>
void
hash_append(HashAlgorithm& h, T* p)
{
    h(&p, sizeof(p));
}
```

rippleLabs

FNV-1A

```
class fnv1a
{
    std::size_t state_ = 14695981039346656037u;
public:
    using result_type = std::size_t;

    void
    operator()(void const* key, std::size_t len) noexcept
    {
        unsigned char const* p = static_cast<unsigned char const*>(key);
        unsigned char const* const e = p + len;
        for (; p < e; ++p)
            state_ = (state_ ^ *p) * 1099511628211u;
    }

    explicit
    operator result_type() noexcept
    {
        return state_;
    }
};
```

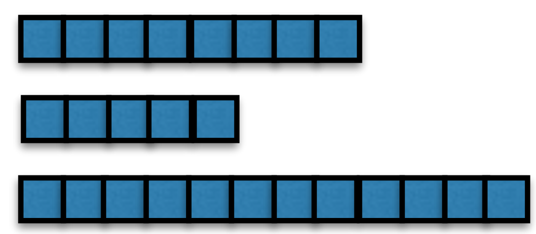
hash_append for primitive types calls this.

rippleLabs

hash_append

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
};
```

- A complicated class like Sale is ultimately made up of scalars located in discontiguous memory.



hash_append

- A complicated class like Sale is ultimately made up of scalars located in discontiguous memory.

```
class Sale
{
    Customer customer_;
    Product  product_;
    Date     date_;
};
```

- hash_append appends each byte to the Hash Algorithm state by recursing down into the data structure to find the scalars.

hash_append

- Every type has a hash_append overload.
- The overload will either call hash_append on its bases and members, or it will send bytes of its memory to the HashAlgorithm.
- No type is aware of the concrete HashAlgorithm type.

To use hash_append

To use hash_append

Initialize the Hash Algorithm

```
HashAlgorithm h;
```

rippleLabs

To use hash_append

Append to the Hash Algorithm

```
HashAlgorithm h;
hash_append(h, t);
```

rippleLabs

To use hash_append

Finalize the Hash Algorithm

```
HashAlgorithm h;
hash_append(h, t);
return static_cast<result_type>(h);
```

rippleLabs

To use hash_append

```
template <class HashAlgorithm = fnv1a>
struct uhash
{
    using result_type = typename HashAlgorithm::result_type;

    template <class T>
    result_type
    operator()(T const& t) const noexcept
    {
        HashAlgorithm h;
        hash_append(h, t);
        return static_cast<result_type>(h);
    }
};
```

Wrap the whole thing up in a conforming hash functor!

rippleLabs

To use hash_append

```
template <class HashAlgorithm = fnv1a>
struct uhash
{
    using result_type = typename HashAlgorithm::result_type;

    template <class T>
    result_type
    operator()(T const& t) const noexcept
    {
        HashAlgorithm h;
        hash_append(h, t);
        return static_cast<result_type>(h);
    }
};
```

Wrap the whole thing up in a conforming hash functor!

```
unordered_set<Customer, uhash<fnv1a>> my_set;
```

rippleLabs

To Change Hashing Algorithms

To Change Hashing Algorithms

```
unordered_set<Sale, uhash< fnv1a > > my_set;
```

rippleLabs

To Change Hashing Algorithms

```
unordered_set<Sale, uhash< SipHash > > my_set;
```

- To change, simply replace the algorithm at the point of use.

To Change Hashing Algorithms

```
unordered_set<Sale, uhash< Spooky > > my_set;
```

- To change, simply replace the algorithm at the point of use.
- Sale does not change!

To Change Hashing Algorithms

```
unordered_set<Sale, uhash< Murmur  > > my_set;
```

- To change, simply replace the algorithm at the point of use.
- Sale does not change!
- Customer does not change!

To Change Hashing Algorithms

```
unordered_set<Sale, uhash< Murmur  > > my_set;
```

- To change, simply replace the algorithm at the point of use.
- Sale does not change!
- Customer does not change!

★ It becomes trivial to experiment with different hashing algorithms to optimize performance, minimize collisions, and secure against attacks.

rippleLabs

To seed a HashAlgorithm

To seed a HashAlgorithm

Initialize the Hash Algorithm

```
HashAlgorithm h{get_seed()};
```

rippleLabs

To seed a HashAlgorithm

Append to the Hash Algorithm

```
HashAlgorithm h{get_seed()};
hash_append(h, t);
```

rippleLabs

To seed a HashAlgorithm

Finalize the Hash Algorithm

```
HashAlgorithm h{get_seed()};
hash_append(h, t);
return static_cast<result_type>(h);
```

rippleLabs

To seed a HashAlgorithm

```
template <class HashAlgorithm = SipHash>
struct seeded_hash
{
    using result_type = typename HashAlgorithm::result_type;

    template <class T>
    result_type
    operator()(T const& t) const noexcept
    {
        HashAlgorithm h{get_seed()};
        hash_append(h, t);
        return static_cast<result_type>(h);
    }
private:
    result_type get_seed();
};
```

Wrap the whole thing up in a conforming hash functor!

```
unordered_set<Customer, seeded_hash<>> my_set;
```

rippleLabs

Set Up Defaults

- It is easy to set everything up to default your favorite hashing algorithm.

Set Up Defaults

- It is easy to set everything up to default your favorite hashing algorithm.

```
template <class T, class Hash = uhash<FavoriteHash>,
          class Pred = std::equal_to<T>,
          class Allocator = std::allocator<T>>
using hash_set = std::unordered_set <T, Hash, Pred, Allocator>;
```

```
hash_set<Customer> set;  // uses FavoriteHash
```

rippleLabs

Details

- There exist traits for optimization purposes:

```
template <class T>
struct is_contiguously_hashable
    : public true_type or false_type
{};
```

- This allows containers like **`tuple`**, **`string`** and **`vector`** to sometimes send all of their data to the hash algorithm at once, which can increase hashing performance, without changing the resulting hash code.

rippleLabs

Details

- There exists a way to write **`hash_append`** for PIMPL types.
- It involves writing a type-erasing **`HashAlgorithm`** *adaptor*.
  - Full source code is of course available.

rippleLabs

Details

- There exist ways to force endian for the input of scalars into the hash algorithms.
- This can be handy when machines with identical layout other than endian need to share hash codes across a network.

Summary

rippleLabs

Summary

- Every type that may be hashed, or participate in a hash computation, must have a `hash_append` overload.

This is the hard part!

rippleLabs

Summary

- Every type that may be hashed, or participate in a hash computation, must have a **`hash_append`** overload.
    - Each type can **`hash_append`** its bases and members.

This is the hard part!

This part is easy.

rippleLabs

Summary

- Every type that may be hashed, or participate in a hash computation, must have a **`hash_append`** overload.
    - Each type can **`hash_append`** its bases and members.
- Known hash algorithms must be adapted to expose their 3 phases: initialization, updating and finalization.

This is the hard part!

This part is easy.

This part is only mildly difficult.

rippleLabs

Summary

- Every type that may be hashed, or participate in a hash computation, must have a **`hash_append`** overload.
  - Each type can **`hash_append`** its bases and members.
- Known hash algorithms must be adapted to expose their 3 phases: initialization, updating and finalization.
- Hash functors initialize a HashAlgorithm, update it with an item to be hashed, and finalize the algorithm.

This is the hard part!

This part is easy.

This part is only mildly difficult.

This part is easy.

rippleLabs

Benefits

rippleLabs

Benefits

- Different hashing algorithms are easily experimented with.

Benefits

- Different hashing algorithms are easily experimented with.
- Hashing algorithms are easily seeded or padded (or not).

Benefits

- Different hashing algorithms are easily experimented with.
- Hashing algorithms are easily seeded or padded (or not).
- Hashing algorithms are computed exactly as their designers intended — there is no hash code combining step.

Benefits

- Different hashing algorithms are easily experimented with.
- Hashing algorithms are easily seeded or padded (or not).
- Hashing algorithms are computed exactly as their designers intended — there is no hash code combining step.
- Hash support (`hash_append`) is implemented for any individual type ***exactly once***. This implementation is good for all hashing algorithms.

rippleLabs

Types Don’t Know

Types Don’t Know

- A type should only know what parts of itself should be presented to a hash algorithm (and in what order).

Types Don't Know

- A type should only know what parts of itself should be presented to a hash algorithm (and in what order).
- A type should not know any specific hash algorithm.

Types Don't Know

- A type should only know what parts of itself should be presented to a hash algorithm (and in what order).
- A type should not know any specific hash algorithm.
- Open source code available at:

https://github.com/HowardHinnant/hash_append

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3980.html

rippleLabs