So You Think You Can Hash

VICTOR CIURA

Cppcon
The C++ Conference

2024
September 15 - 20

So You Think You Can Hash

CppCon

September 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
Rust Tooling @ Microsoft

Abstract

Hashing is crucial for efficient data retrieval and storage. This presentation delves into computing hashes for aggregated user-defined types and experimenting with various hash algorithms. We will explore the essentials of hash functions and their properties, techniques for hashing complex user-defined types, and customizing std::hash for specialized needs.

Additionally, we (re)introduce a framework for experimenting with and benchmarking different hash algorithms. This will allow easy switching of hashing algorithms used by complex data structures, enabling easy comparisons.

Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code. Type designers can create their hash support just once, without worrying about what hashing algorithm should be used.

You will gain practical insights and tools to implement, customize, and evaluate hash functions in C++, enhancing software performance and reliability.

About me

Advanced Installer

Clang Power Tools

Oxidizer SDK

Visual C++

Rust Tooling

🐦 @ciura_victor
🐘 @ciura_victor@hachyderm.io
🦋 @ciuravictor.bsky.social

Motivation

Hashing is crucial for efficient data retrieval and storage.
This exploration delves into computing hashes for aggregated user-defined types and experimenting with various hash algorithms.

We will explore the essentials of hash functions and their properties, techniques for hashing complex user-defined types, and customizing std hash for specialized needs.

Motivation

A hashing "framework" for:

- easy experimenting and benchmarking with different hash algorithms
- easy swapping of hashing algorithms (later on)
- hashing complex aggregated user-defined types
- enabling easy comparisons of hashing techniques

Goals

Goals

Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

Goals

Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

Type designers (developers) can create their hash support just once, without worrying about what hashing algorithm should be used.

Goals

Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

Type designers (developers) can create their hash support just once, without worrying about what hashing algorithm should be used.

We'll try to gain practical insights and mechanisms to implement, customize, and evaluate hash functions, enhancing software performance and reliability.

Primer

Most programming languages offer some kind of associative containers.

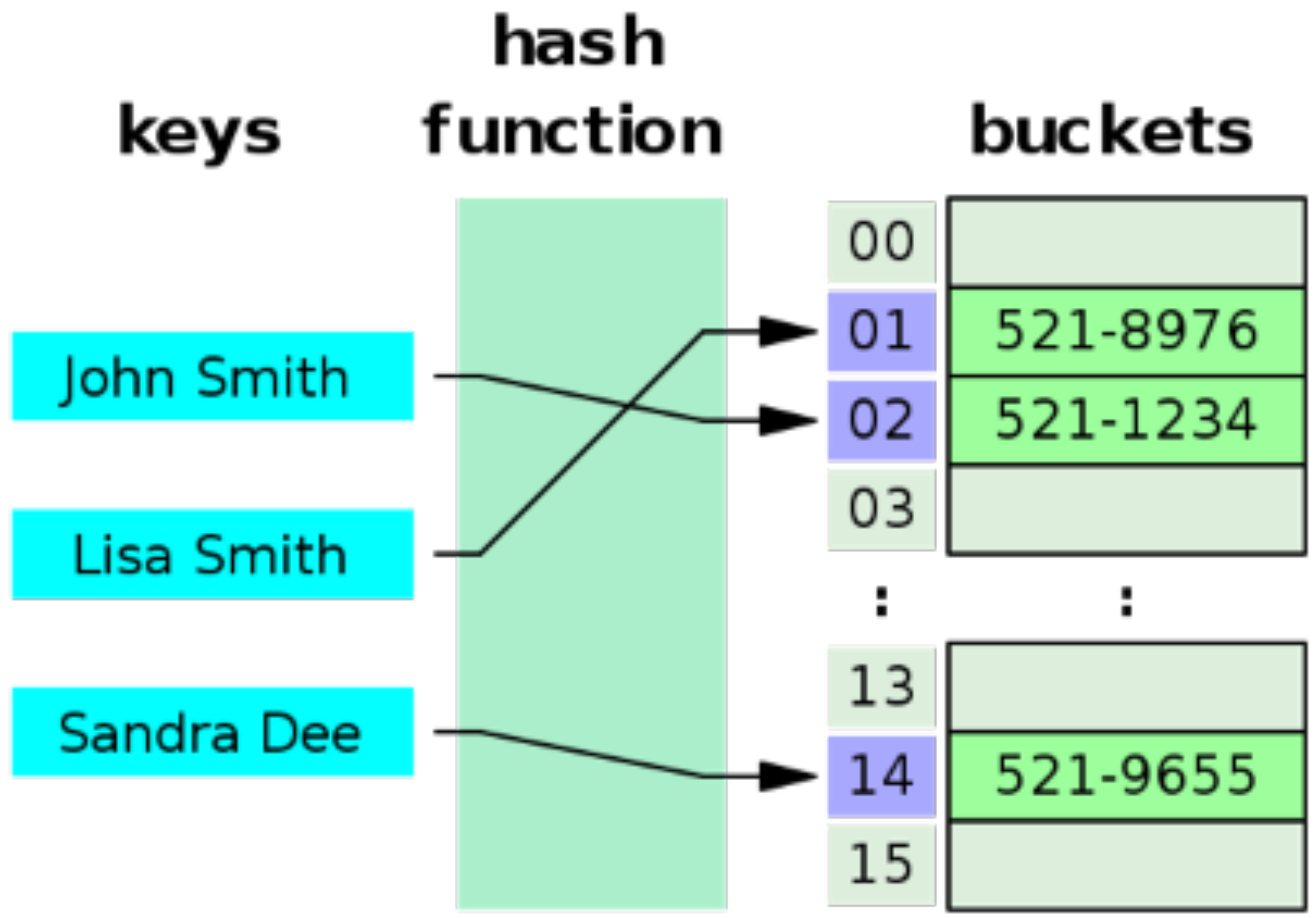
They may be called differently: maps, dictionaries, hash-maps, unordered-maps, hash-tables, etc.

Hash Functions & Hash Tables

A hash function is any function that can be used to map data of arbitrary size to data of *fixed size* (hash code).

Hash functions are used in hash tables, to quickly locate a data record given its search key.

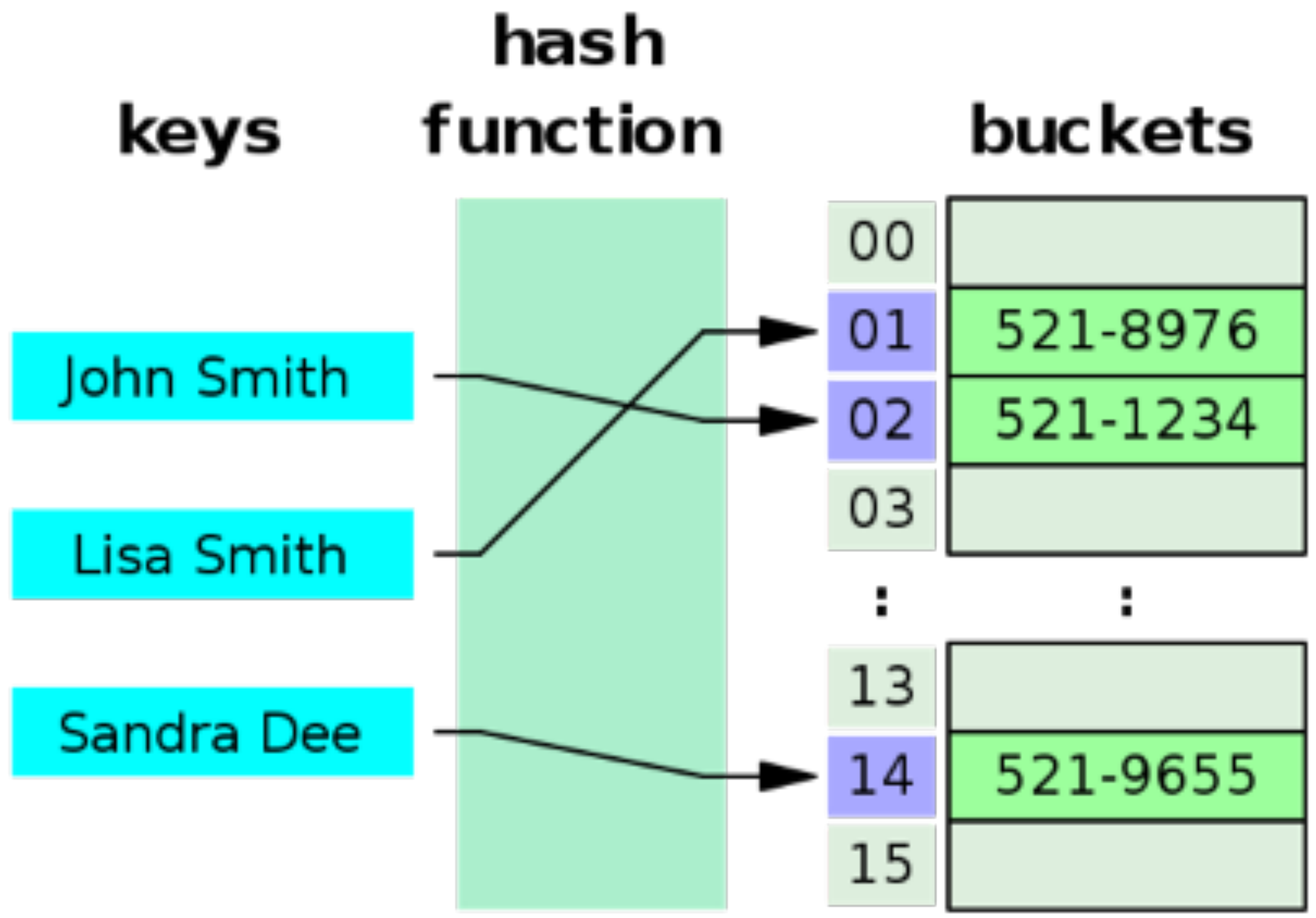
The hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored/found.



Domain

The domain of a hash function (the set of possible keys) is larger than its range (the number of different table indices), and so it will map several different keys to the same index.

Each slot (bucket) of a hash table is associated with a set of records, rather than a single record.



Properties

Determinism

A hash procedure must be deterministic — meaning that for a given input value it must always generate the same hash value.

Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range.

That is, every hash value in the output range should be generated with roughly the same probability.

Properties

Defined Range

It is often desirable that the output of a hash function have fixed size.

If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array (eg. hash tables).

Properties

Non-invertible

In cryptographic applications, hash functions are typically expected to be *practically* non-invertible, meaning that it is not realistic to reconstruct the input datum from its hash value alone, without spending great amounts of computing time.

Questions

Questions

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

Questions

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

Questions

- How should one combine hash codes from your 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 aggregate function, how would you change it for a type built out of several data members (that are not primitive types)?

Questions

- How should one combine hash codes from your 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 aggregate function, how would you change it for a type built out of several data members (that are not primitive types)?
- How to separate concerns: hash algorithms from the aggregation of the digest (combine) and from the collection type itself (HashMap, BTreeMap, etc)?

Hashing Composite Types

Let's assume we want to store some custom struct into a hash map, but we can't use any unique/identifier field as key into the container (no UUID, no unique string). So, we need a means of inserting such structure as key:

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
};

std::unordered_map<Customer, Records> customer_records;
```

Hashing Composite Types

Let's assume we want to store some custom struct into a hash map, but we can't use any unique/identifier field as key into the container (no UUID, no unique string). So, we need a means of inserting such structure as key:

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
};

std::unordered_map<Customer, Records> customer_records;
```

Instead of the plain:

```
std::unordered_map<String, CustomerRecords> customer_records;
std::unordered_map<Uuid,   CustomerRecords> customer_records;
```

Hashing Composite Types

How does one hash this type?

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
  //...
};
```

std::hash

`std::hash<Key>`

- Accepts a single parameter of type Key
- Returns a value of type size_t that represents the hash value of the parameter
- Does not throw exceptions when called
- If `k1 == k2` ➡️ `hash<Key>()(k1) == hash<Key>()(k2)`
- If `k1 != k2` ➡️ the probability that `hash<Key>()(k1) == hash<Key>()(k2)`

  should be very small, approaching `1.0/numeric_limits<size_t>::max()`

```
std::size_t h1 = std::hash<std::string>{}(firstName);
```

std::hash

Specializations for *basic* types:

```
template< class T > struct hash<T*>;

template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<short>;
template<> struct hash<unsigned short>;
template<> struct hash<int>;
template<> struct hash<unsigned int>;
template<> struct hash<long>;
template<> struct hash<long long>;
template<> struct hash<unsigned long>;
template<> struct hash<unsigned long long>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;
```

Specializations for *library* types:

```
std::hash<std::string>
std::hash<std::wstring>
std::hash<std::unique_ptr>
std::hash<std::shared_ptr>
std::hash<std::bitset>
//...
```

Hashing Composite Types

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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);


  }
};
```

Is this a good hash strategy?

Hashing Composite Types

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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?
  }
};
```

🤔 Is this a good hash strategy?

Hashing Composite Types

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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?
  }
};
```

Is this a good hash strategy?

What if we wanted to use another hash algorithm?

hash_combine()

hash_combine()

But what to do with these hash codes now?

hash_combine()

But what to do with these hash codes now?

hash_combine()

But what to do with these hash codes now?

How should we combine them to obtain a unified hash representing our whole structure?

hash_combine()

But what to do with these hash codes now?

How should we combine them to obtain a unified hash representing our whole structure?

hash_combine()

But what to do with these hash codes now?

How should we combine them to obtain a unified hash representing our whole structure?

Believe it or not, there are such numerical algorithms for combining hash codes and retaining the desirable properties of a good hasher.

hash_combine()

But what to do with these hash codes now?

How should we combine them to obtain a unified hash representing our whole structure?

Believe it or not, there are such numerical algorithms for combining hash codes and retaining the desirable properties of a good hasher.

```
template <class T>
inline void hash_combine(std::size_t & seed, const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

hash_combine()

```
template <class T>
inline void hash_combine(std::size_t & seed, const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

hash_combine()

```
template <class T>
inline void hash_combine(std::size_t & seed, const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

The magic number is supposed to be 32 “random” bits:

- each is equally likely to be 0 or 1
- with no simple correlation between the bits

A common way to find a pattern of such bits is to use the binary expansion of an *irrational number*.

In this case, that number is the *reciprocal* of the golden ratio:

```
φ = (1 + sqrt(5)) / 2
2^32 / φ = 0x9e3779b9
```

hash_combine()

Such solutions while working in most cases, are not without problems, both in terms of numerical/mathematical soundness (? any hash algorithm), but also in terms of flexibility/composition of the code using them.

hash_combine()

Such solutions while working in most cases, are not without problems, both in terms of numerical/mathematical soundness (? any hash algorithm), but also in terms of flexibility/composition of the code using them.

```
std::size_t hash_code() const
{
  std::size_t customer_hash = 0; // is this a good seed?

  hash_combine(customer_hash, firstName);
  hash_combine(customer_hash, lastName);
  hash_combine(customer_hash, age);
  //...

  return customer_hash;
}
```

hash_combine()

Such solutions while working in most cases, are not without problems, both in terms of numerical/mathematical soundness (? any hash algorithm), but also in terms of flexibility/composition of the code using them.

```
std::size_t hash_code() const
{
  std::size_t customer_hash = 0; // is this a good seed?

  hash_combine(customer_hash, firstName);
  hash_combine(customer_hash, lastName);
  hash_combine(customer_hash, age);
  //...

  return customer_hash;
}
```

hash algorithm hidden inside

hash_combine()

```
std::size_t customer_hash = 0; // is this a good seed?

hash_combine(customer_hash, firstName);
hash_combine(customer_hash, lastName);
hash_combine(customer_hash, age);
//...
```

hash_combine()

```
std::size_t customer_hash = 0; // is this a good seed?

hash_combine(customer_hash, firstName);
hash_combine(customer_hash, lastName);
hash_combine(customer_hash, age);
//...
```

The end result is that the algorithm is polluted by the combine step.

Is this a good hash strategy? Probably not.

hash_combine()

```
std::size_t customer_hash = 0; // is this a good seed?

hash_combine(customer_hash, firstName);
hash_combine(customer_hash, lastName);
hash_combine(customer_hash, age);
//...
```

The end result is that the algorithm is polluted by the combine step.

Is this a good hash strategy? Probably not.

What if we wanted to use another hash algorithm?

The numerical solution (0x9e3779b9) for combining hash codes from std::hash might not be sound for other hash algorithms. 🤔

Unpack `std::hash`

But what's inside `std::hash`?

What's the algorithm used?

Unpack `std::hash`

But what's inside `std::hash`?

What's the algorithm used?

`FNV–1A`

Fowler-Noll-Vo hash was designed for fast hash-table and checksum use (not crypto).

But what's inside `std::hash`?

What's the algorithm used?

FNV-1A

Fowler-Noll-Vo hash was designed for fast hash-table and checksum use (not crypto).

```
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;
}
```

Unpack `std::hash`

But what's inside `std::hash`?

What's the algorithm used?

FNV–1A

Fowler-Noll-Vo hash was designed for fast hash-table and checksum use (not crypto).

```
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;
}
```

wikipedia.org/wiki/Fowler–Noll–Vo_hash_function

FNV-1A

```
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;
}
```

FNV_offset_basis

FNV_prime

We can easily apply such a hash function to obtain hash codes for all common types (primitive or std) we might have in our class and even do that *recursively*, if we have multiple-level aggregation.

FNV-1A

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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));


  }
};
```

FNV-1A

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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); // FNV1-A combine? Can we reuse this?
  }
};
```

FNV-1A

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
// ...
  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); // FNV1-A combine? Can we reuse this?
  }
};
```

That didn't get us far... 🤷‍♂️

Our algorithm is still “polluted” by the combine step...

Insights

If we analyze most hashing algorithms in common use:

- `FNV1a`
- `SipHash`
- `Spooky`
- `Murmur`
- `CityHash`
- etc.

we notice that they all share some common anatomy in their implementation.

Anatomy of a Hash Function

1. **Initialize** internal state
2. **Consume** bytes into internal state
3. **Finalize** internal state to result type (usually size_t)

Anatomy of a Hash Function

```
std::size_t fnv1a(void const * key, std::size_t len)
{
  std::size_t h = 14695981039346656037u;  ⇐ initialize internal state

  // consume bytes into internal state:
  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;  ⇐ finalize internal state to size_t
}
```

Anatomy of a Hash Function

In this particular case (FNV1a), the **Initialize** and **Finalize** steps are trivial, but for other hashing algorithms, these might be much more involved and very costly (runtime).

So, we want to make sure that even if idempotent with regards to the end hash code, we don't execute the initialization and finalize steps more than once, when we compute the hash code of a complex/nested aggregate structure.

Repackaging

What we need to do is to repackage the algorithm, in a generic way
(to work with all types of hashers), to make the 3 stages above separately accessible:

1. Init / construction of the hasher
2. Write overloads for primitive/std types (*append* to the hash)
3. Finalize function -> size_t

Repackaging

What we need to do is to repackage the algorithm, in a generic way
(to work with all types of hashers), to make the 3 stages above separately accessible:

1. Init / construction of the hasher
2. Write overloads for primitive/std types (*append* to the hash)
3. Finalize function -> size_t

This technique ensures that:

- we no longer need to have a combine step
- we're using the same hash algorithm for the entire data structure
  (no special "glue" for *intermediate* hash codes)

Repackaging

The salient idea here is that you let some "other" piece of code construct and finalize the hashing algorithm.

`Customer` struct only **appends** to the state of the hasher.
Indeed, hashers need to become *stateful*, for this pattern to work.

Repackaging

```
class fnv1a
{
  std::size_t h = 14695981039346656037u;     ⇐ initialize internal state
public:

  // consume bytes into internal state
  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 size_t() noexcept  ⇐ finalize internal state to size_t
  {
    return h;
  }
};
```

made the 3 stages separately accessible

Hashing Composite Types

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(), lastName.size());
    hasher(&age, sizeof(age));

    return static_cast<std::size_t>(hasher);
  }
};
```

Notice anything missing? 🙋

Hashing Composite Types

Notice anything missing? 🙋

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(), lastName.size());
    hasher(&age, sizeof(age));

    return static_cast<std::size_t>(hasher); ⬅️ no more hash_combine() !!!
  }
};
```

Hashing Composite Types

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(), lastName.size());
    hasher(&age, sizeof(age));

    return static_cast<std::size_t>(hasher);  ⬅️ no more hash_combine() !!!
  }
};
```

Notice anything missing? 🙋

Now we are using a "pure" FNV-1A algorithm for the entire data structure. (no more "glue" hash code)

Swap the Hasher

This clean separation/repackaging of the 3 phases of hashing also allows great flexibility in swapping the hasher algorithm without the need to touch the data model and how each field recursively contributes to the overall digest (append/write).

🔶 The same technique can be used with almost every existing hashing algorithm, eg. `FNV1a, SipHash, Spooky, Murmur, CityHash`.

Hashing Composite Types

Let's move one more level: nested aggregate 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();

    return hash_combine(h1, h2, h3);
  }
};
```

Hashing Composite Types

Let's move one more level: nested aggregate 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();

    return hash_combine(h1, h2, h3); ⬅️ OMG, it’s back 😱
  }
};
```

Hashing Composite Types

Let's move one more level: nested aggregate 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();

    return hash_combine(h1, h2, h3); ⬅️ OMG, it’s back 😱
  }
};
```

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

hash_append()

Remember our Customer?

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(),  lastName.size());
    hasher(&age,             sizeof(age));

    return static_cast<std::size_t>(hasher);
  }
};
```

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(),  lastName.size());
    hasher(&age,             sizeof(age));

    return static_cast<std::size_t>(hasher);
  }
};
```

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(),  lastName.size());
    hasher(&age,             sizeof(age));

    return static_cast<std::size_t>(hasher);
  }
};
```

Let some other piece of code construct and finalize the fnv1a hash.
Customer should only **append** to the state of fnv1a hasher.

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  friend void hash_append(fnv1a & hasher, const Customer & c)
  {
    ...............

    hasher(c.firstName.data(), c.firstName.size());
    hasher(c.lastName.data(),  c.lastName.size());
    hasher(&c.age,             sizeof(c.age));

    ...........................................
  }
};
```

Let some other piece of code construct and finalize the fnv1a hash.
Customer should only **append** to the state of fnv1a hasher.

Hashing Composite Types

Back to our nested aggregate types:

```
class Sale
{
  Customer customer;
  Product  product;
  Date     date;

public:

  friend void hash_append(fnv1a & hasher, const Sale & s)
  {
    hash_append(hasher, s.customer);
    hash_append(hasher, s.product);
    hash_append(hasher, s.date);
  }
};
```

Types can "recursively" build upon one another's `hash_append()` to build up *state* in **fnv1a** object.

hash_append(🐢🐢🐢)

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  friend void hash_append(fnv1a & hasher, const Customer & c)
  {
    hash_append(hasher, c.firstName);
    hash_append(hasher, c.lastName);
    hash_append(hasher, c.age);
  }
};
```

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

=> simplified & uniform interface

Abstracting the algorithm

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  template<class HashAlgorithm>
  friend void hash_append(HashAlgorithm & hasher, const Customer & c)
  {
    hash_append(hasher, c.firstName);
    hash_append(hasher, c.lastName);
    hash_append(hasher, c.age);
  }
};
```

If all hash algorithms use a uniform interface, we can swap any hasher into our data type.

Primitives

For primitive types that are contiguously hashable
we can just send their bytes to the hash algorithm, in `hash_append()`.

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

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

Recipe 🧾

Even a complicated class is ultimately made up of scalars, located in discontiguous memory.

`hash_append()` appends each byte to the HashAlgorithm state by "*recursing down*" into the aggregated data structure to find the scalars.

Recipe 🧾

Even a complicated class is ultimately made up of scalars, located in discontiguous memory.

`hash_append()` appends each byte to the HashAlgorithm state by "*recursing down*" into the aggregated data structure to find the scalars.

Steps:

- Every type has a `hash_append()` overload
- Each overload will either call `hash_append()` on its bases and members, or it will send bytes of its memory representation to the HashAlgorithm (scalars)
- No type is aware of the concrete HashAlgorithm implementation

Recipe 🧾

Even a complicated class is ultimately made up of scalars, located in discontiguous memory.

`hash_append()` appends each byte to the HashAlgorithm state by "*recursing down*" into the aggregated data structure to find the scalars.

Just the salient parts of types

Steps:

- Every type has a `hash_append()` overload
- Each overload will either call `hash_append()` on its bases and members, or it will send bytes of its memory representation to the HashAlgorithm (scalars)
- No type is aware of the concrete HashAlgorithm implementation

Tough bits...

There are some areas of debatable design considerations, wrt to hashing:

- `std::optional`
  - should have a *presence indicator* in the hash?
  - should we consider optional as an either `0` or `1` size container of T?

- `std::variant`
  - how should we encode the type discriminant?

I want to chat with you about some of these things... 🤔

Example

```
{
  SomeHashAlgorithm hasher;

  hash_append(hasher, my_type);

  return static_cast<size_t>(hasher);
}
```

Example

```
{
  SomeHashAlgorithm hasher;

  hash_append(hasher, my_type);

  return static_cast<size_t>(hasher);
}
```

OK, but how do I stick this into a `std::unordered_set/map` ?

GenericHash

Just wrap the whole thing up in a conforming hash function object:

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

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

```
std::unordered_set<Customer, GenericHash<fnv1a>> my_set;
```

Hash algorithms... everywhere

Hash algorithms

It becomes trivial to experiment with different hashing algorithms, to benchmark & optimize performance, minimize collisions, tune for the input data, etc.

```
std::unordered_set<Sale, GenericHash<fnv1a>> my_set;

std::unordered_set<Sale, GenericHash<SipHash>> my_set;

std::unordered_set<Sale, GenericHash<Spooky>> my_set;

std::unordered_set<Sale, GenericHash<Murmur>> my_set;

std::unordered_set<Sale, GenericHash<CityHash>> my_set;

...
```

ISO-terica...

Paper	Date (last rev)	Title	Discussion
N3333	2012-01-13	Hashing User-Defined Type in C++1y	N3333 Discussion
N3573	2013-03-10	Heterogenous extensions to unordered containers	
N3730	2013-08-28	Specializations and namespaces	
N3876	2014-01-19	Convenience Functions to Combine Hash Values	N3876 and N3898 Discussion
N3898	2014-01-20	Hashing and Fingerprinting	N3898 Discussion
N3983	2014-05-07	Hashing tuple-like types	
N3980	2014-05-24	Types Don't Know #	
N3339	2015-04-09	Message Digest Library for C++	
P0029r0	2015-09-21	A Unified Proposal for Composable Hashing	P0029 Discussion
P0199r0	2016-02-11	Default Hash	
P0513r0	2016-11-10	Poisoning the Hash	D0513 Discussion
P0599r1	2017-03-02	noexcept for Hash Functions	D0599 Discussion
P0809r0	2017-10-12	Comparing Unordered Containers	
P0814r2	2018-02-12	hash_combine() Again	P0814 Discussion
P0549r7	2020–02–17	Adjuncts to std::hash	

ISO-terica...

- There were plenty of hashing-related papers in WG21.
- Some of these try to build on prior work. Some bring forth new ideas.
- Subsequent papers do not necessarily address the discussion point from previous work.
- The discussion points brought up do not necessarily represent a consensus view.

📖 gist.github.com/dietmarkuehl/file-lets-hash-things-over-md

ISO-terica...

Document number: P0029R0
Date: 2015-09-21
Project: Programming Language C++, Library Evolution Working Group
Reply to: Geoff Romer <gromer@google.com>, Chandler Carruth <chandlerc@google.com>

A Unified Proposal for Composable Hashing

Document number: N3980

Howard E. Hinnant
Vinnie Falco
John Bytheway
2014-05-24

Types Don't Know

N. Josuttis: P0814R2: hash_combine() Again, Rev2

Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P0814R2**
Date:	2018-02-12
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Audience:	LEWG, LWG
Prev. Version:	P0814R1

Adjuncts to `std::hash`

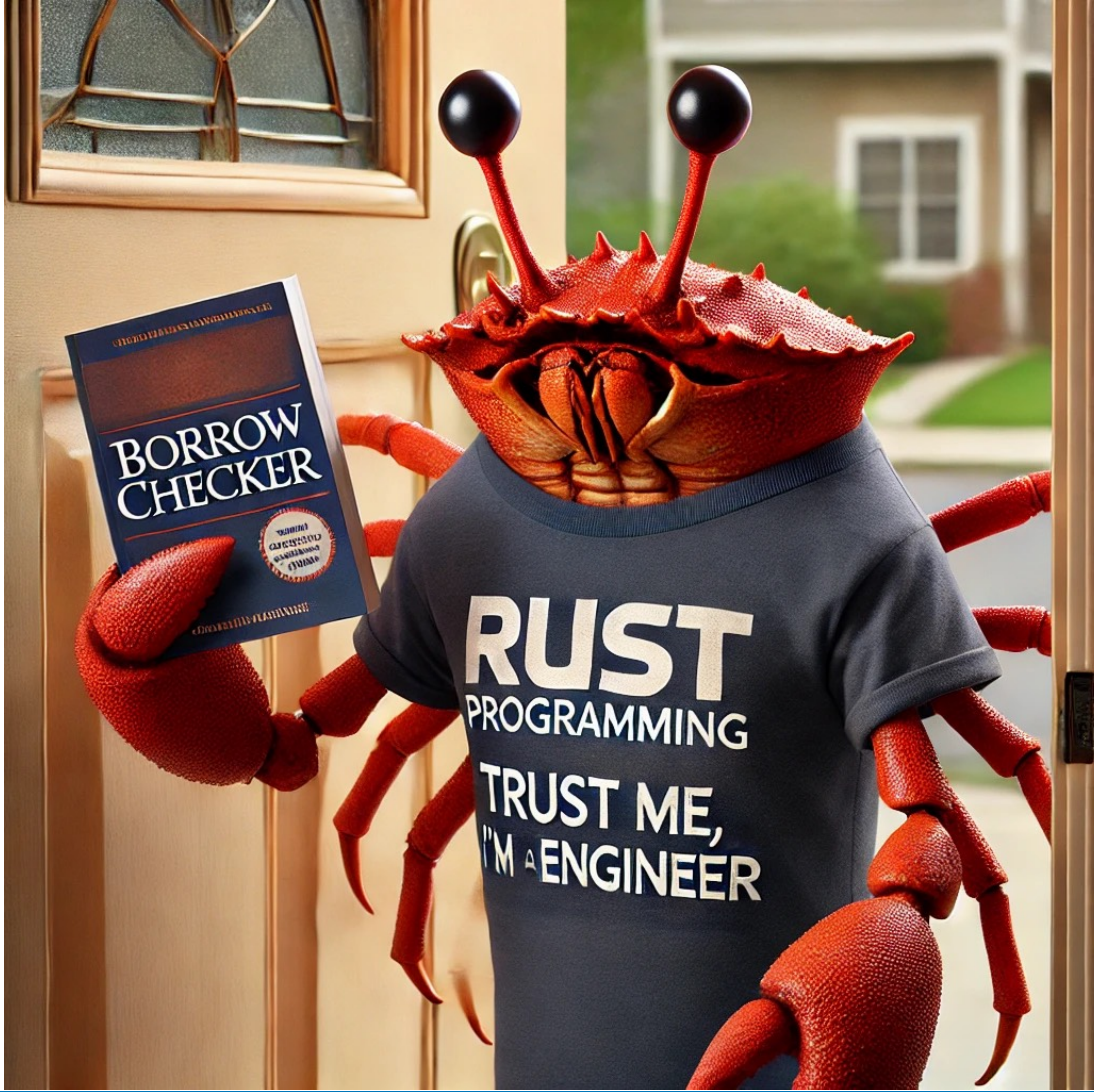
Document #:	WG21 P0549R7
Date:	2020–02–17
Project:	JTC1.22.32 Programming Language C++
Audience:	LWG
Audience:	~~LEWG~~ done ⇒ LWG
Reply to:	Walter E. Brown <`webrown.cpp@gmail.com`>

Rust ❤️ C++

I'm not here to:

- convert anyone to 🦀 Rust
- start any language wars
- *"sell the Rust snake oil"*
- tell you to RiiR

So, don't throw 🍅



Trait std::hash::Hash

```
// Required method
fn hash<H>(&self, state: &mut H)
    where H: Hasher;
```

This function feeds this value type into the given Hasher.

This is the append method, that contributes to the overall hash digest by recursively calling `hash()` on all constituents of the structure.

```
impl Hash for Customer {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.first_name.hash(state);
        self.last_name.hash(state);
        self.age.hash(state);
        self.premium.hash(state);
    }
}
```

#[derive(Hash)]

You can derive Hash with #[derive(Hash)] if all fields implement Hash.

The resulting hash will be the combination of the values from calling hash on each field.

```
#[derive(Hash)]
struct Customer {
    first_name: String,
    last_name: String,
    age: i32,
    premium: bool,
}
```

#[derive(Hash)]

You can derive Hash with `#[derive(Hash)]` if all fields implement Hash.

The resulting hash will be the combination of the values from calling hash on each field.

```
#[derive(Hash)]
struct Customer {
    first_name: String,
    last_name: String,
    age: i32,
    premium: bool,
}
```

When implementing Equality for a type, we want to make sure equal values map to equal hash codes.

This might not be always true, if not all members participate in the equality relationship.

➡️

`#[derive(PartialEq, Eq, Hash)]` ensures that property is upheld.

Trait std::hash::Hash

This trait has implementors for almost all std types. See the complete list here.

Eg.

```
impl Hash for str {
    #[inline]
    fn hash<H: Hasher>(&self, state: &mut H) {
        state.write_str(self);
    }
}

impl Hash for String {
    #[inline]
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        (**self).hash(hasher)  <== falls back on the &str impl
    }
}
```

Trait std::hash::Hasher

And this brings us to the actual Hasher object, that will implement a particular algorithm.

```
    // Required methods
    fn finish(&self) -> u64;
    fn write(&mut self, bytes: &[u8]);

    // Provided methods (many helpers)
    fn write_u8(&mut self, i: u8) { ... }
    fn write_i32(&mut self, i: i32) { ... }
...
    fn write_str(&mut self, s: &str) { ... }
```

This is the part of the hashing infra that provides the protocol for a particular Hasher implementation – that holds the algorithm for the hasher.

Trait std::hash::Hasher

Instances of Hasher usually represent **state** that is changed while hashing data, by "appending" to the hash digest and ultimately ensuring that the algorithm finalization step is executed.

Most of the time, Hasher instances are used in conjunction with the Hash trait.

```
let mut hasher = DefaultHasher::new();

hasher.write_u32(1989);
hasher.write_u8(11);
hasher.write_i64(1729);
hasher.write_str("Foo");

println!("Hash is {:x}", hasher.finish());
```

Trait std::hash::Hasher

There is a potentially brittle aspect of this design:
➡️ the order of subsequent `write()` calls cannot be checked/enforced, eg. for aggregated structs.

Thus, to produce the same hash value, Hash implementations must ensure for equivalent items that exactly the same sequence of calls is made – the same methods with the same parameters in the same order.

Trait std::hash::Hasher

There is a potentially brittle aspect of this design:
➡️ the order of subsequent `write()` calls cannot be checked/enforced, eg. for aggregated structs.

Thus, to produce the same hash value, Hash implementations must ensure for equivalent items that exactly the same sequence of calls is made – the same methods with the same parameters in the same order.

> If your type is implementing Hash, you generally do not need to call these `write()` functions directly, as the `[impl Hash]` does, so you should prefer that instead.

Trait std::hash::Hasher

The Rust std library provides a couple of implementors for this trait:

Trait std::hash::Hasher

The Rust std library provides a couple of implementors for this trait:

- `RandomState` - is the ***default*** state for std HashMap types.

Trait std::hash::Hasher

The Rust std library provides a couple of implementors for this trait:

- `RandomState` - is the *default* state for std HashMap types.
- `DefaultHasher`
  - the internal algorithm is not specified, and so it and its hashes should not be relied upon over releases
  - a general-purpose hashing algorithm (SipHasher13): it runs at a good speed (competitive with Spooky and City) and permits strong keyed hashing
  - the default Hasher used by `RandomState`

Trait std::hash::Hasher

The Rust std library provides a couple of implementors for this trait:

- `RandomState` - is the *default* state for std HashMap types.
- `DefaultHasher`
  - the internal algorithm is not specified, and so it and its hashes should not be relied upon over releases
  - a general-purpose hashing algorithm (SipHasher13): it runs at a good speed (competitive with Spooky and City) and permits strong keyed hashing
  - the default Hasher used by `RandomState`
- `SipHasher` [deprecated]

Trait std::hash::Hasher

The Rust std library provides a couple of implementors for this trait:

- `RandomState` - is the *default* state for std HashMap types.
- `DefaultHasher`
  - the internal algorithm is not specified, and so it and its hashes should not be relied upon over releases
  - a general-purpose hashing algorithm (SipHasher13): it runs at a good speed (competitive with Spooky and City) and permits strong keyed hashing
  - the default Hasher used by `RandomState`
- `SipHasher` [deprecated]
- `Adler32` – a typical Adler-32 checksum

Trait std::hash::BuildHasher

A trait for creating *instances* of `Hasher`.

A `BuildHasher` is typically used (eg. by HashMap) to create Hashers for each key such that they are hashed *independently* of one another, since Hashers contain state (digest).

```
fn build_hasher(&self) -> Self::Hasher;
```

Trait std::hash::BuildHasher

A trait for creating *instances* of `Hasher`.

A `BuildHasher` is typically used (eg. by HashMap) to create Hashers for each key such that they are hashed *independently* of one another, since Hashers contain state (digest).

```
fn build_hasher(&self) -> Self::Hasher;
```

For each instance of `BuildHasher`, the `Hashers` created should be *identical*.

That is, if the same stream of bytes is fed into each hasher, the same output will also be generated:

Trait std::hash::BuildHasher

A trait for creating *instances* of `Hasher`.

A `BuildHasher` is typically used (eg. by HashMap) to create Hashers for each key such that they are hashed *independently* of one another, since Hashers contain state (digest).

```
fn build_hasher(&self) -> Self::Hasher;
```

For each instance of `BuildHasher`, the `Hashers` created should be *identical*.

That is, if the same stream of bytes is fed into each hasher, the same output will also be generated:

```
let s = RandomState::new();
let mut hasher_1 = s.build_hasher();
let mut hasher_2 = s.build_hasher();
hasher_1.write_u32(8128);
hasher_2.write_u32(8128);
assert_eq!(hasher_1.finish(), hasher_2.finish());
```

std::hash::BuildHasherDefault

The standard way to create a default `BuildHasher` instance for types that implement `Hasher` and `Default`.

```
#[derive(Default)]
struct FancyHasher;

impl Hasher for FancyHasher {
    fn write(&mut self, bytes: &[u8]) {
       // hashing algorithm (append/digest)
    }

    fn finish(&self) -> u64 {
        // hashing algorithm (finalization step)
    }
}

type FancyBuildHasher = BuildHasherDefault<FancyHasher>;
let hash_map = HashMap::<Customer, Records, FancyBuildHasher>::default();
```

Complex Aggregates

```
struct Sale {
    customer: Customer,
    product:  Product,
    date:     Date,
}
```

Types can recursively build upon one another's `hash()` to build up state in Hasher object.

`hash()` appends each byte to the Hasher *state* by recursing down into the data structure, to find the scalars (plain types). -- just for the salient parts of the data

Complex Aggregates

```
impl Hash for Sale {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.customer.hash(state); // deep traversal to Customer hashing
        self.product.hash(state);  // deep traversal to Product hashing
        self.date.hash(state);     // deep traversal to Date hashing
    }
}

impl Hash for Date {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.year.hash(state); // deep traversal stops on trivial type (u32)
        self.month.hash(state);// deep traversal stops on trivial type (u32)
        self.day.hash(state);  // deep traversal stops on trivial type (u32)
    }
}
```

Complex Aggregates

If no customization is needed (how the type needs to contribute to the hash digest), the simplest path is to derive:

```
#[derive(Hash, Eq, PartialEq)]
struct Sale { }

// Define the HashMap with the default hasher
let mut sales_map: HashMap<Sale, u64> = HashMap::new();
sales_map.insert(Sale::new(...), 1500);
```

Complex Aggregates

If no customization is needed (how the type needs to contribute to the hash digest), the simplest path is to derive:

```
#[derive(Hash, Eq, PartialEq)]
struct Sale { }

// Define the HashMap with the default hasher
let mut sales_map: HashMap<Sale, u64> = HashMap::new();
sales_map.insert(Sale::new(...), 1500);


// Define the HashMap with a custom hasher (eg. SipHasher)
type SipHasherMap<K, V> = HashMap<K, V, BuildHasherDefault<SipHasher>>;
let mut sales_map: SipHasherMap<Sale, u64> = HashMap::default();

sales_map.insert(Sale::new(...), 1500);
```

Conclusion

`.finalize()`

We introduced a hashing "framework" for:

- easy experimenting and benchmarking with different hash algorithms
- easy swapping of hashing algorithms (later on)
- hashing complex aggregated user-defined types
- enabling easy comparisons of hashing techniques

Conclusion

.finalize()

We introduced a hashing "framework" for:

- easy experimenting and benchmarking with different hash algorithms
- easy swapping of hashing algorithms (later on)
- hashing complex aggregated user-defined types
- enabling easy comparisons of hashing techniques

```
std::unordered_set<Sale, GenericHash<fnv1a>> my_set;

std::unordered_set<Sale, GenericHash<SipHash>> my_set;

std::unordered_set<Sale, GenericHash<Spooky>> my_set;

std::unordered_set<Sale, GenericHash<Murmur>> my_set;

std::unordered_set<Sale, GenericHash<CityHash>> my_set;
```

Goals

`.digest()`

✔

✔

✔

✔

Goals

`.digest()`

✔ Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

✔

✔

✔

Goals

`.digest()`

✔ Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

✔ Type designers (developers) can create their hash support just once, without worrying about what hashing algorithm should be used.

✔

✔

Goals

`.digest()`

- ✓ Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.
- ✓ Type designers (developers) can create their hash support just once, without worrying about what hashing algorithm should be used.
- ✓ We gained practical insights and mechanisms to implement, customize, and evaluate hash functions, enhancing software performance and reliability.
- ✓

Goals

`.digest()`

- ✓ Hash algorithm designers can concentrate on designing better hash algorithms, with little worry about how these new algorithms can be incorporated into existing code.

- ✓ Type designers (developers) can create their hash support just once, without worrying about what hashing algorithm should be used.

- ✓ We gained practical insights and mechanisms to implement, customize, and evaluate hash functions, enhancing software performance and reliability.

- ✓ We want to enforce a clear separation: no type should be aware of the concrete HashAlgorithm to be used with it, rather only be concerned with how it contributed to the digest (which underlying parts).

So You Think You Can Hash

CppCon

September 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
Rust Tooling @ Microsoft