So You Think You Can

Victor Ciura - Technical Lead, Advanced Installer
http://www.advancedinstaller.com
CAPHYON

April 2017

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 <u>quickly</u> 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.

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 <u>different keys</u> to the <u>same index</u>.

=> Each slot (**bucket**) of a hash table is associated with a <u>set of records</u>, rather than a single record.

Hash Function 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.

❖ **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).

❖ **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

- How should one <u>combine</u> 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?

How does one hash this class?

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

std::hash<Key>

Defined in header <functional>

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

- 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 and k2 are equal => std::hash<Key>()(k1) == std::hash<Key>()(k2)
- If k1 and k2 are different, the ***probability*** that std::hash<Key>()(k1) == std::hash<Key>()(k2) should be very small, approaching 1.0/std::numeric_limits<size_t>::max()

std::hash<Key>

Standard 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>;
```

Standard 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>
//...
```

```cpp
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?
  }
};
```

Is this a good hash strategy?

What if we wanted to use another hash algorithm?

boost::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**

http://stackoverflow.com/questions/35985960/c-why-is-boosthash-combine-the-best-way-to-combine-hash-values

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;
}
```

The FNV hash was designed for fast hash-table and checksum use (not cryptography).

https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

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

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
}
```

Repackaging this algorithm
to make the three stages separately accessible

```
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;
  }
};
```

```cpp
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() !!!
  }
};
```

- The same technique can be used with almost every existing hashing algorithm.

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

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();

    return hash_combine(h1, h2, h3); // OMG, he’s back :(
  }
};
```

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

hash_append()

Proposal for C++ ISO standardization by:

Howard Hinnant, Vinnie Falco, John Bytheway

Document number: N3980 / 2014-05-24

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); // no more hash_combine() !!!
  }
};
```

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* fnv1a.
Customer only *appends* to the state of fnv1a.

hash_append()

```
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

hash_append() / 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.

hash_append() / Primitives

For **primitive types** that are ***contiguously hashable***
we can just send their <u>bytes</u> to the hash algorithm in hash_append().

Eg.

```
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));
}
```

hash_append()

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 data structure to find the scalars.

Prerequisites:

- 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.

How to use hash_append()

```
HashAlgorithm hasher;

hash_append(hasher, my_type);

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

Wrap the whole thing up in a conforming hash functor

```
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);
  }
};
```

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

Change Hashing Algorithms

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

unordered_set<Sale, GenericHash<SipHash>> my_set;

unordered_set<Sale, GenericHash<Spooky>> my_set;

unordered_set<Sale, GenericHash<Murmur>> my_set;

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

It becomes trivial to experiment with different hashing algorithms to optimize performance, minimize collisions.

Exploring string hash tables
<code walk-through>

What we want:

"A hash table mapping string keys (case-insensitive) to some custom data type."

Starting point:

```
template<
  class Key,
  class T,
  class Hash     = std::hash<Key>,
  class KeyEqual = std::equal_to<Key>, ⇐ Why is this part of the interface? Why not Key::operator==()
  class Alloc    = std::allocator< std::pair<const Key, T> >
>
class std::unordered_map;
```

What we need:

- A custom hash functor for case-insensitive strings
- A custom comparator functor, to compare strings ignoring character case

Exploring string hash tables
<code walk-through>

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHash
{
  using HashedType = StringType;

  size_t operator()(const HashedType & aStr) const
  {
    std::hash<HashedType> hasher;  ⇐ we can use any hashing algorithm
    return hasher(aStr);
  }

  bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
  {
    return aStr1 < aStr2;
  }

  struct KeyEquality
  {
    bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
    {
      return aStr1 == aStr2;
    }
  };
};
```

Exploring string hash tables
<code walk-through>

```
typedef BasicStringHash<char>     StringHash;
typedef BasicStringHash<wchar_t>  StringHashW;
```

Eg.

```
        std::unordered_map<wstring, TYPE, StringHashW, StringHashW::KeyEquality>
```

Exploring string hash tables
<code walk-through>

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHashI
{
  using HashedType = StringType;

  size_t operator()(const HashedType & aStr) const
  {
    // make a lower-case copy of the input string
    HashedType lowerStr(aStr);
    ToLower(const_cast<Type *>(lowerStr.c_str()));

    std::hash<HashedType> hasher;
    return hasher(lowerStr);
  }

  bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
  {
    return CompareI(aStr1, aStr2) < 0;
  }

  //...

};
```

Case-insensitive hashes

Exploring string hash tables
<code walk-through>

Case-insensitive hashes

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHashI
{
  //...

  struct KeyEquality
  {
    bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
    {
      return CompareI(aStr1, aStr2) == 0;
    }
  };

private:
  static void ToLower(char    * aStr) {  ::CharLowerA(aStr);  }
  static void ToLower(wchar_t * aStr) {  ::CharLowerW(aStr);  }

  static int CompareI(const string & aStr1, const string & aStr2)
  {  return ::lstrcmpiA(aStr1.c_str(), aStr2.c_str());  }

  static int CompareI(const wstring & aStr1, const wstring & aStr2)
  {  return ::lstrcmpiW(aStr1.c_str(), aStr2.c_str());  }

};
```

Exploring string hash tables
<code walk-through>

```
typedef BasicStringHashI<char>     StringHashI;
typedef BasicStringHashI<wchar_t>  StringHashWI;
```

Eg.

```
std::unordered_map<wstring, TYPE, StringHashWI, StringHashWI::KeyEquality>
```

Case-insensitive hashes

Exploring string hash tables
<code walk-through>

Special type of case-insensitive hash: **file-paths** hash map.

What we want:

```
std::unordered_map<FilePath, TYPE, FilePathHash, FilePathHash::KeyEquality>
```

Where **FilePath** encapsulates a std::**wstring** plus file specific methods.

What issues do we have with regular **StringHashWI** for file paths ?

```
std::unordered_map<FilePath, TYPE, StringHashWI, StringHashWI::KeyEquality>
```

Research Topic for You

Optimal file-path (case-insensitive)
hash functor for std::unordered_map<>

Details* to come in a separate Handout **PDF** document (soon).

* problem statement (industrial usage context, current issues, goals)
data domain (usual input, training data sets)
operating constraints
benchmarking strategy (baseline, measurement markers)