What's the best hashing algorithm to use on a stl string when using hash_map?

Asked 16 years, 3 months ago Modified 7 years, 11 months ago

Viewed 55k times

49

I've found the standard hashing function on VS2005 is painfully slow when trying to achieve high performance look ups. What are some good examples of fast and efficient hashing algorithms that should void most collisions?

c++ windows performance stl hash

Share

Improve this question

Follow

edited Sep 19, 2008 at 0:07

Dark Shikari

8,009 ● 4 ● 28 ● 38

asked Sep 18, 2008 at 23:58

PiNoYBoY82

1,648 ● 2 ● 14 ● 17

25 The following has a good set of general purpose hash functions, you should try them against your data set, some may outperform others based on collisions: partow.net/programming/hashfunctions/index.html
– Matthieu N. Oct 24, 2009 at 9:00

1 Possible duplicate of Good Hash Function for Strings
– M.J. Rayburn Sep 2, 2017 at 1:04

11 Answers

Sorted by: Highest score (default)

66

I worked with Paul Larson of Microsoft Research on some hashtable implementations. He investigated a number of string hashing functions on a variety of datasets and found that a simple multiply by 101 and add loop worked surprisingly well.

```
unsigned int
hash(
    const char* s,
    unsigned int seed = 0)
{
    unsigned int hash = seed;
    while (*s)
    {
        hash = hash * 101  +  *s++;
    }
    return hash;
}
```

Share Improve this answer Follow

edited May 1, 2015 at 0:55

answered Sep 20, 2008 at 8:46

George V. Reilly
16.3k ● 7 ● 45 ● 39

1 Hey George. I tried the code in the hash-benchmark I've postet in my answer. Nice find. It does not excel in performance or collisions, but it always gives consistent results. Seems like it's a good and cheap candidate for general purpose string-hashing. – Nils Pipenbrinck Sep 23, 2008 at 1:10

1 But this works only for small length strings. For large cases, it overflows majority of the time. – Soumajyoti Sarkar Jan 3, 2013 at 14:13

13 Soumajyoti, the overflow doesn't matter. Most hash functions overflow. The point is that you get a decent mix of bits in the low-order 32-bits. – George V. Reilly Jan 4, 2013 at 23:12

2 That resembles the Java implementation, but it uses 31 instead of 101. – Jorge Galvão Mar 12, 2014 at 10:25

How can we find the hash of substring, given we have already calculated the hashes of all prefixes of given String? Example, consider `s = "hello"`, the hashes of all prefixes of s are: `[104, 10605, 1071213, 108192621, 2337520240]` How to quickly find hash of substring `hel`? – Sai Suman Chitturi Sep 3, 2021 at 9:21

19

From some old code of mine:

```
/* magic numbers from http://www.isthe.com/chongo/tech
static const size_t InitialFNV = 2166136261U;
static const size_t FNVMultiple = 16777619;

/* Fowler / Noll / Vo (FNV) Hash */
size_t myhash(const string &s)
```

```
{
    size_t hash = InitialFNV;
    for(size_t i = 0; i < s.length(); i++)
    {
        hash = hash ^ (s[i]);       /* xor  the low 8
        hash = hash * FNVMultiple;  /* multiply by the
    }
    return hash;
}
```

Its fast. Really freaking fast.

Share Improve this answer Follow

answered Sep 19, 2008 at 0:01

Dark Shikari
8,009 ● 4 ● 28 ● 38

5 it may be fast, but its probably one of the WORST hash functions ever invented. – Matthieu N. Mar 1, 2010 at 23:08

6 @Matthieu: Why? Many duplicates? Do you have any references where I can read more about it? – Albert Oct 17, 2012 at 15:12

1 @Albert: `^` is transitive, which is bad. `FNVMultiple` is not prime, which is bad. `InitialFNV` isn't prime either, which may or may not be bad, I'm uncertain. – Mooing Duck Jun 8, 2015 at 15:35

@MooingDuck FNVMultiple seems to be a prime number. – bysreg Jul 31, 2017 at 2:08

1 16777619 is a (proven) prime. 2166136261 is (proven) composite (failed sprp test base 2). primes.utm.edu/curios/includes/primetest.php – Nick Feb 21, 2018 at 10:15

7

That always depends on your data-set.

I for one had surprisingly good results by using the CRC32 of the string. Works very good with a wide range of different input sets.

Lots of good CRC32 implementations are easy to find on the net.

Edit: Almost forgot: This page has a nice hash-function shootout with performance numbers and test-data:

http://smallcode.weblogs.us/ <-- further down the page.

Share Improve this answer Follow

answered Sep 18, 2008 at 23:59

Nils Pipenbrinck
86.2k ● 33 ● 155 ● 223

7

Boost has an boost::hash library which can provides some basic hash functions for most common types.

Share Improve this answer Follow

answered Sep 19, 2008 at 0:01

David Pierre
9,535 ● 4 ● 42 ● 32

I've use the Jenkins hash to write a Bloom filter library, it has great performance.

6

Details and code are available here: http://burtleburtle.net/bob/c/lookup3.c

This is what Perl uses for its hashing operation, fwiw.

Share Improve this answer Follow

answered Sep 19, 2008 at 0:24

SquareCog
19.6k ● 8 ● 51 ● 63

Also look at the spooky hash which is an improvement on Jenkins – Soren Oct 15, 2015 at 16:39

6

If you are hashing a fixed set of words, the best hash function is often a perfect hash function. However, they generally require that the set of words you are trying to hash is known at compile time. Detection of keywords in a lexer (and translation of keywords to tokens) is a common usage of perfect hash functions generated with tools such as gperf. A perfect hash also lets you replace `hash_map` with a simple array or `vector`.

If you're not hashing a fixed set of words, then obviously this doesn't apply.

Share Improve this answer Follow

edited Feb 6, 2010 at 16:06

answered Sep 19, 2008 at 3:13

bk1e
24.3k ● 6 ● 57 ● 65

2

One classic suggestion for a string hash is to step through the letters one by one adding their ascii/unicode values to an accumulator, each time multiplying the accumulator by a prime number. (allowing overflow on the hash value)

```
  template <> struct myhash{};

  template <> struct myhash<string>
    {
    size_t operator()(string &to_hash) const
      {
```

```
      const char * in = to_hash.c_str();
      size_t out=0;
      while(NULL != *in)
        {
        out*= 53; //just a prime number
        out+= *in;
        ++in;
        }
      return out;
      }
    };

  hash_map<string, int, myhash<string> > my_hash_map;
```

It's hard to get faster than that without throwing out data. If you know your strings can be differentiated by only a few characters and not their whole content, you can do faster.

You might try caching the hash value better by creating a new subclass of basic_string that remembers its hash value, if the value gets calculated too often. hash_map should be doing that internally, though.

Share Improve this answer Follow

edited Sep 19, 2008 at 1:30

answered Sep 19, 2008 at 0:18

Brian
8,813 ● 5 ● 29 ● 30

1 Yoda condition alert! Apart from that this is similar to the Larson algorithm (I noticed this was posted earlier!).
– Helge Klein Aug 17, 2014 at 11:51

2

I did a little searching, and funny thing, Paul Larson's little algorithm showed up here http://www.strchr.com/hash_functions as having the least collisions of any tested in a number of conditions, and it's very fast for one that it's unrolled or table driven.

Larson's being the simple multiply by 101 and add loop above.

Share Improve this answer Follow

answered Feb 20, 2012 at 2:41

Josh S
147 ● 1 ● 7

2

Python 3.4 includes a new hash algorithm based on SipHash. PEP 456 is very informative.

Share Improve this answer Follow

answered Mar 19, 2014 at 18:29

George V. Reilly
16.3k ● 7 ● 45 ● 39

1 I run some benchmarks and SipHash looks very good
– David Soroko Apr 24, 2015 at 20:53

From Hash Functions all the way down:

1

> MurmurHash got quite popular, at least in game developer circles, as a “general hash function”.
>
> It’s a fine choice, but let’s see later if we can generally do better. Another fine choice, especially if you know more about your data than “it’s gonna be an unknown number of bytes”, is to roll your own (e.g. see Won Chun’s replies, or Rune’s modified xxHash/Murmur that are specialized for 4-byte keys etc.). If you know your data, always try to see whether that knowledge can be used for good effect!

Without more information I would recommend MurmurHash as a general purpose non-cryptographic hash function. For small strings (of the size of the average identifier in programs) the very simple and famous djb2 and FNV are very good.

> Here (data sizes < 10 bytes) we can see that the ILP smartness of other algorithms does not get to show itself, and the super-simplicity of FNV or djb2 win in performance.

djb2

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33
 
    return hash;
}
```

FNV-1

```
hash = FNV_offset_basis
for each byte_of_data to be hashed
     hash = hash × FNV_prime
     hash = hash XOR byte_of_data
return hash
```

FNV-1A

```
hash = FNV_offset_basis
for each byte_of_data to be hashed
     hash = hash XOR byte_of_data
     hash = hash × FNV_prime
return hash
```

A note about security and availability

Hash functions can make your code vulnerable to denial-of-service attacks. If an attacker is able to force your server to handle too many collisions, your server may not be able to cope with requests.

Some hash functions like MurmurHash accept a seed that you can provide to drastically reduce the ability of attackers to predict the hashes your server software is generating. Keep that in mind.

Share Improve this answer Follow

edited Dec 24, 2016 at 17:27

answered Dec 24, 2016 at 15:08

felipecrv
2,019 ● 2 ● 17 ● 18

@FelixSFD I just improved the answer. – felipecrv Dec 24, 2016 at 17:28

0

If your strings are on average longer than a single cache line, but their length+prefix are reasonably unique, consider hasing just the length+first 8/16 characters. (The length is contained in the std::string object itself and therefore cheap to read)

Share Improve this answer Follow

answered Sep 19, 2008 at 11:14

MSalters
179k ● 11 ● 164 ● 368