```
//-----------------------------------------------------------------------------
// MurmurHash3 was written by Austin Appleby, and is placed in the public
// domain. The author hereby disclaims copyright to this source code.

// Note - The x86 and x64 versions do _not_ produce the same results, as the
// algorithms are optimized for their respective platforms. You can still
// compile and run any of them on any platform, but your performance with the
// non-native version will be less than optimal.

#include "MurmurHash3.h"

//-----------------------------------------------------------------------------
// Platform-specific functions and macros

// Microsoft Visual Studio

#if defined(_MSC_VER)

#define FORCE_INLINE	__forceinline

#include <stdlib.h>

#define ROTL32(x,y) _rotl(x,y)
#define ROTL64(x,y) _rotl64(x,y)

#define BIG_CONSTANT(x) (x)

// Other compilers

#else	// defined(_MSC_VER)

#define	FORCE_INLINE inline __attribute__((always_inline))

inline uint32_t rotl32 ( uint32_t x, int8_t r )
{
  return (x << r) | (x >> (32 - r));
}

inline uint64_t rotl64 ( uint64_t x, int8_t r )
{
  return (x << r) | (x >> (64 - r));
}

#define	ROTL32(x,y)	rotl32(x,y)
#define ROTL64(x,y)	rotl64(x,y)

#define BIG_CONSTANT(x) (x##LLU)

#endif // !defined(_MSC_VER)
```

```
//-----------------------------------------------------------------------------
// Block read - if your platform needs to do endian-swapping or can only
// handle aligned reads, do the conversion here

FORCE_INLINE uint32_t getblock32 ( const uint32_t * p, int i )
{
  return p[i];
}

FORCE_INLINE uint64_t getblock64 ( const uint64_t * p, int i )
{
  return p[i];
}

//-----------------------------------------------------------------------------
// Finalization mix - force all bits of a hash block to avalanche

FORCE_INLINE uint32_t fmix32 ( uint32_t h )
{
  h ^= h >> 16;
  h *= 0x85ebca6b;
  h ^= h >> 13;
  h *= 0xc2b2ae35;
  h ^= h >> 16;

  return h;
}

//----------

FORCE_INLINE uint64_t fmix64 ( uint64_t k )
{
  k ^= k >> 33;
  k *= BIG_CONSTANT(0xff51afd7ed558ccd);
  k ^= k >> 33;
  k *= BIG_CONSTANT(0xc4ceb9fe1a85ec53);
  k ^= k >> 33;

  return k;
}

//-----------------------------------------------------------------------------

void MurmurHash3_x86_32 ( const void * key, int len,
                          uint32_t seed, void * out )
{
  const uint8_t * data = (const uint8_t*)key;
  const int nblocks = len / 4;

  uint32_t h1 = seed;
```

```
  const uint32_t c1 = 0xcc9e2d51;
  const uint32_t c2 = 0x1b873593;

  //----------
  // body

  const uint32_t * blocks = (const uint32_t *)(data + nblocks*4);

  for(int i = -nblocks; i; i++)
  {
    uint32_t k1 = getblock32(blocks,i);

    k1 *= c1;
    k1 = ROTL32(k1,15);
    k1 *= c2;

    h1 ^= k1;
    h1 = ROTL32(h1,13);
    h1 = h1*5+0xe6546b64;
  }

  //----------
  // tail

  const uint8_t * tail = (const uint8_t*)(data + nblocks*4);

  uint32_t k1 = 0;

  switch(len & 3)
  {
  case 3: k1 ^= tail[2] << 16;
  case 2: k1 ^= tail[1] << 8;
  case 1: k1 ^= tail[0];
          k1 *= c1; k1 = ROTL32(k1,15); k1 *= c2; h1 ^= k1;
  };

  //----------
  // finalization

  h1 ^= len;

  h1 = fmix32(h1);

  *(uint32_t*)out = h1;
}

//-----------------------------------------------------------------------------

void MurmurHash3_x86_128 ( const void * key, const int len,
```

```
                          uint32_t seed, void * out )
{
  const uint8_t * data = (const uint8_t*)key;
  const int nblocks = len / 16;

  uint32_t h1 = seed;
  uint32_t h2 = seed;
  uint32_t h3 = seed;
  uint32_t h4 = seed;

  const uint32_t c1 = 0x239b961b;
  const uint32_t c2 = 0xab0e9789;
  const uint32_t c3 = 0x38b34ae5;
  const uint32_t c4 = 0xa1e38b93;

  //----------
  // body

  const uint32_t * blocks = (const uint32_t *)(data + nblocks*16);

  for(int i = -nblocks; i; i++)
  {
    uint32_t k1 = getblock32(blocks,i*4+0);
    uint32_t k2 = getblock32(blocks,i*4+1);
    uint32_t k3 = getblock32(blocks,i*4+2);
    uint32_t k4 = getblock32(blocks,i*4+3);

    k1 *= c1; k1  = ROTL32(k1,15); k1 *= c2; h1 ^= k1;

    h1 = ROTL32(h1,19); h1 += h2; h1 = h1*5+0x561ccd1b;

    k2 *= c2; k2  = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

    h2 = ROTL32(h2,17); h2 += h3; h2 = h2*5+0x0bcaa747;

    k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;

    h3 = ROTL32(h3,15); h3 += h4; h3 = h3*5+0x96cd1c35;

    k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

    h4 = ROTL32(h4,13); h4 += h1; h4 = h4*5+0x32ac3b17;
  }

  //----------
  // tail

  const uint8_t * tail = (const uint8_t*)(data + nblocks*16);

  uint32_t k1 = 0;
```

```
uint32_t k2 = 0;
uint32_t k3 = 0;
uint32_t k4 = 0;

switch(len & 15)
{
case 15: k4 ^= tail[14] << 16;
case 14: k4 ^= tail[13] << 8;
case 13: k4 ^= tail[12] << 0;
         k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

case 12: k3 ^= tail[11] << 24;
case 11: k3 ^= tail[10] << 16;
case 10: k3 ^= tail[ 9] << 8;
case  9: k3 ^= tail[ 8] << 0;
         k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;

case  8: k2 ^= tail[ 7] << 24;
case  7: k2 ^= tail[ 6] << 16;
case  6: k2 ^= tail[ 5] << 8;
case  5: k2 ^= tail[ 4] << 0;
         k2 *= c2; k2  = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

case  4: k1 ^= tail[ 3] << 24;
case  3: k1 ^= tail[ 2] << 16;
case  2: k1 ^= tail[ 1] << 8;
case  1: k1 ^= tail[ 0] << 0;
         k1 *= c1; k1  = ROTL32(k1,15); k1 *= c2; h1 ^= k1;
};

//----------
// finalization

h1 ^= len; h2 ^= len; h3 ^= len; h4 ^= len;

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

h1 = fmix32(h1);
h2 = fmix32(h2);
h3 = fmix32(h3);
h4 = fmix32(h4);

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

((uint32_t*)out)[0] = h1;
((uint32_t*)out)[1] = h2;
((uint32_t*)out)[2] = h3;
((uint32_t*)out)[3] = h4;
```

```
}

//-----------------------------------------------------------------------------

void MurmurHash3_x64_128 ( const void * key, const int len,
                           const uint32_t seed, void * out )
{
  const uint8_t * data = (const uint8_t*)key;
  const int nblocks = len / 16;

  uint64_t h1 = seed;
  uint64_t h2 = seed;

  const uint64_t c1 = BIG_CONSTANT(0x87c37b91114253d5);
  const uint64_t c2 = BIG_CONSTANT(0x4cf5ad432745937f);

  //----------
  // body

  const uint64_t * blocks = (const uint64_t *)(data);

  for(int i = 0; i < nblocks; i++)
  {
    uint64_t k1 = getblock64(blocks,i*2+0);
    uint64_t k2 = getblock64(blocks,i*2+1);

    k1 *= c1; k1  = ROTL64(k1,31); k1 *= c2; h1 ^= k1;

    h1 = ROTL64(h1,27); h1 += h2; h1 = h1*5+0x52dce729;

    k2 *= c2; k2  = ROTL64(k2,33); k2 *= c1; h2 ^= k2;

    h2 = ROTL64(h2,31); h2 += h1; h2 = h2*5+0x38495ab5;
  }

  //----------
  // tail

  const uint8_t * tail = (const uint8_t*)(data + nblocks*16);

  uint64_t k1 = 0;
  uint64_t k2 = 0;

  switch(len & 15)
  {
  case 15: k2 ^= ((uint64_t)tail[14]) << 48;
  case 14: k2 ^= ((uint64_t)tail[13]) << 40;
  case 13: k2 ^= ((uint64_t)tail[12]) << 32;
  case 12: k2 ^= ((uint64_t)tail[11]) << 24;
  case 11: k2 ^= ((uint64_t)tail[10]) << 16;
```

```
  case 10: k2 ^= ((uint64_t)tail[ 9]) << 8;
  case  9: k2 ^= ((uint64_t)tail[ 8]) << 0;
           k2 *= c2; k2  = ROTL64(k2,33); k2 *= c1; h2 ^= k2;

  case  8: k1 ^= ((uint64_t)tail[ 7]) << 56;
  case  7: k1 ^= ((uint64_t)tail[ 6]) << 48;
  case  6: k1 ^= ((uint64_t)tail[ 5]) << 40;
  case  5: k1 ^= ((uint64_t)tail[ 4]) << 32;
  case  4: k1 ^= ((uint64_t)tail[ 3]) << 24;
  case  3: k1 ^= ((uint64_t)tail[ 2]) << 16;
  case  2: k1 ^= ((uint64_t)tail[ 1]) << 8;
  case  1: k1 ^= ((uint64_t)tail[ 0]) << 0;
           k1 *= c1; k1  = ROTL64(k1,31); k1 *= c2; h1 ^= k1;
  };

  //----------
  // finalization

  h1 ^= len; h2 ^= len;

  h1 += h2;
  h2 += h1;

  h1 = fmix64(h1);
  h2 = fmix64(h2);

  h1 += h2;
  h2 += h1;

  ((uint64_t*)out)[0] = h1;
  ((uint64_t*)out)[1] = h2;
}

//-----------------------------------------------------------------------------
```