Sample hash functions .Most of these functions hash C-style strings (string), although a few work with integers. Not all of them have been extensively tested to be sure they generate good hash values.

All of the functions fall into one of two classes: integer hashers and string hashers. The integer hashers all take a signed integer key; the string hashers take a C-style string (string).

All functions return an unsigned integer suitable for use as a hash-table index. With one exception (hashStringBase256), this integer can be larger than the size of the hash table, so the caller must reduce the integer modulo the table size.

The functions are:

hashStringCRC	Uses a method sometimes called "modified CRC". This is a simple, fast algorithm that generates good hash values. It is closely related to the hash algorithm used in ispell.
hashStringBase256	Treats a string as a very large base-256 number and returns that number modulo the table size. This is the only function that accepts the table size as an argument, and the only one that guarantees that its result will be in the range 0 to tableSize - 1. It should produce good results but has not been tested extensively.
hashPJW	Uses a method recommended by Aho, Sethi, and Ullman's book on compilers. Seems to generate pretty good values.
hashIntegerMultiply	Uses a multiplicative method. Not extensively tested.
hashStringBUZ	Uses a randomization method that is intended to produce better hash values. Seems to generate pretty good values, but doesn't do much better with the ispell dictionary than hashStringCRC.
hashIntegerBUZ	Uses the same randomization method by treating an integer as a 4-character string.
hashIntegerAbs	Returns the absolute value of the hashed integer. Not extensively tested. Should work well if the integer keys are well distributed and the table size is prime.
hashStringPreiss	Uses the method recommended in Preiss's book on data structures. Not extensively tested. Probably produces poor hash values for long

	strings that differ only in the leading characters, especially if those characters are transpositions of each other.
hashStringWeiss1	Uses the method recommended in the first edition of Weiss's book on data structures. Not extensively tested. Produces poor hash values for long strings that differ only in the leading characters.
hashStringWeiss2	Uses the method recommended in the second edition of Weiss's book on data structures. Not extensively tested. Produces poor hash values for long strings that differ only in the leading characters.

```
/*
 * Table of Contents:
 */

unsigned int   hashStringCRC(const string& key);
                                // Hash a string using the fake-CRC XOR method
unsigned int   hashStringBase256(const string& key, unsigned int tableSize);
                                // Hash a string by interpreting it in base 256
unsigned int   hashPJW(const string& key);
                                // Hash a string with Aho/Sethi/Ullman's method
unsigned int   hashIntegerMultiply(int key);
                                // Hash an integer multiplicatively
unsigned int   hashStringBUZ(const string& key);
                                // Hash a string with the BUZ method
unsigned int   hashIntegerBUZ(int key);
                                // Hash an integer with the BUZ method
static unsigned int hashBUZ(const unsigned char* key, int keyLength);
                                // Helper function for BUZ hashing
unsigned int   hashIntegerAbs(int key);
                                // Hash an integer by taking the absolute value
unsigned int   hashStringPreiss(const string& key);
                                // Hash a string using Preiss's method
unsigned int   hashStringWeiss(const string& key);
                                // Hash a string using Weiss's method

/*
 * Constants used by the various hash methods.  Constants marked with
 * a "32-BIT" comment are predicated on the assumption of a 32-bit
 * word, and won't work on a 16-bit or 64-bit machine.
 */
static const unsigned int BYTE_WIDTH = 8;
                                // Number of bits in a byte
static const unsigned int WORD_WIDTH = sizeof (int) * BYTE_WIDTH;
```

```
                                // Number of bits in a machine word (int), used
                                // ..by several algorithms.

static const unsigned int CRC_HASH_SHIFT = 5;
                                // How much to shift hash by, per char hashed

static const unsigned int PJW_HASH_SHIFT = 4;
                                // How much to shift hash by, per char hashed
static const unsigned int PJW_HASH_RIGHT_SHIFT = 24;
                                // Right-shift amount, if top 4 bits NZ
                                // 32-BIT
static const unsigned int PJW_HASH_MASK = 0xf0000000;
                                // Mask for extracting top 4 bits
                                // 32-BIT

static const unsigned int PREISS_HASH_SHIFT = 6;
                                // How much to shift hash by, per char hashed
static const unsigned int PREISS_HASH_MASK =
                            ~0U << (WORD_WIDTH - PREISS_HASH_SHIFT);
                                // Mask for extracting top bits

static const unsigned int WEISS_HASH_SHIFT = 5;
                                // How much to shift hash by, per char hashed
static const unsigned int WEISS_HASH_MULTIPLIER = 37;
                                // How much to multiply by, per char hashed

/*
 * Constants for the BUZ algorithm.  BUZ_INIT is the initial value of
 * the hash function.  The lookup table is used to widen and randomize
 * the bytes of the key.  All values were gotten from /dev/random on a
 * Linux box.
 */
const unsigned int BUZINIT = 0x7b4402a2;
static unsigned int BUZ_Table[256] =
  {
  0xc70bb269, 0x13299943, 0xe9cee5e1, 0x32119a79,
  0xc2365851, 0x169ee8e3, 0x094c1ed8, 0x58e1d4e0,
  0x96eb1762, 0xfe296797, 0x89082f47, 0x27d5078d,
  0x8ebb9de0, 0x14df49e0, 0x38d21c63, 0x1f5b2770,
  0xaa6b0150, 0x7b0b656e, 0x98d37bc2, 0x4d7e85ec,
  0x987910e2, 0xb8cbac89, 0xa3f664a3, 0xeca3003c,
  0x7b364137, 0xb2a6edae, 0x5ef72906, 0x64a9e7b7,
  0x28cd6520, 0xca3c72df, 0x57398ce2, 0x8db893d4,
  0x0a5995cd, 0x2d109fb4, 0x0491162f, 0xb3488737,
  0x6fc4eb03, 0x9903cb21, 0xe82ff831, 0xb03ff8e5,
  0x054836ca, 0x5335e6f8, 0x01396c2a, 0xf9000899,
  0x03ed9d63, 0x2bf6946b, 0x9097fa8b, 0xacd8dfc7,
  0x8488b8a6, 0x0e39cd2e, 0xac1a4517, 0xcd49e035,
```

0xe98b7e7b, 0xd3571502, 0xd602805e, 0xe7143cfe,
0x46db0a6b, 0x0a4c9ebe, 0x4e2e1ca7, 0x3040fc62,
0xe8818c02, 0x37155e7b, 0xe44ba138, 0x43cacdd4,
0x53d986ba, 0xdd4dca35, 0x0f680f71, 0x6c1a551e,
0x74263e95, 0xcfc4f5d5, 0x37b8ef45, 0xc00ac71d,
0x3b059e0d, 0x208bc754, 0x41335fbe, 0x785a0ffc,
0x189f024f, 0xd669c2d8, 0xe1b20f87, 0xba2550da,
0x10167369, 0x85fad38f, 0x97d20e4e, 0x5bc0da5e,
0x80799570, 0x93eb4058, 0x139042a6, 0x40b34bf6,
0x15c21dfa, 0x8f852660, 0xa3d20fb3, 0x3d175cf9,
0x792441a8, 0xdc5e71b5, 0x925f6350, 0x66e8d08b,
0xc4606b59, 0x85d8b88c, 0x1ea4f459, 0x664f62bc,
0x77407de3, 0x73d158ca, 0xb76ab172, 0xe9ed1aeb,
0x93dc2009, 0xeb9da6ac, 0x3d26cf05, 0x675132bc,
0xc29196fe, 0x2a62486f, 0x914e75e1, 0xa1c31883,
0x1c28291c, 0xc73c668c, 0xf4ac07e6, 0x87c9a9ac,
0xb7196ea7, 0x67cb7fa2, 0x55987797, 0x29ce38ea,
0x427361b3, 0x5b5667a6, 0x68a72fb0, 0xcef8235a,
0xd06e8f5b, 0x4d3633f5, 0x214d3a19, 0xbd09ec15,
0x5c61c24b, 0x3928573c, 0x26083ab8, 0x857a5dee,
0x3203e50c, 0x52a1a713, 0xa8270ee2, 0xdfb643a9,
0x7797c1f3, 0x0f8ddc9f, 0x9368de21, 0x638ebd4e,
0xd91808d7, 0x28ce69b8, 0xe424b0ce, 0xfe52fdef,
0x89126c74, 0xdb5f3d91, 0xba488f47, 0x2b15cdb8,
0xa517b0f9, 0x53950632, 0x1159546a, 0xe50f65a3,
0x5f26b5d1, 0x68a3a955, 0xc2b78ea1, 0x49c33701,
0x45457aee, 0xd49b550a, 0x244379b8, 0xec826af5,
0x4fa6e0c9, 0xd4633425, 0x82f0bd85, 0xc23ccc2e,
0xac73e11a, 0xdc94b283, 0x13e59bb2, 0x23b4880e,
0x1d295c45, 0xef67488c, 0x6b74149e, 0xdf90d4ac,
0xfc6e65a9, 0x406a3734, 0x86999303, 0xc73e7180,
0x3c8a0b31, 0x75fa9249, 0xaca5e0e2, 0x4d0cc60d,
0x4b174606, 0x836fb602, 0x4f9fc83a, 0xe16477a7,
0xda1506a9, 0x905b28b7, 0x4229f5c2, 0xdf4c9144,
0x731888e9, 0x2e37421f, 0x0c67c385, 0x44a2e520,
0xfe3ee655, 0x92547582, 0x9525f4d4, 0xdaa8caf8,
0xa25bd583, 0x0e315733, 0xd35fea29, 0xc9cfaa0f,
0xc6bdf7e9, 0xa48b4e01, 0xfd30ffe0, 0xd0f63421,
0x2b84803d, 0xe1b368a4, 0xbae5daa5, 0x4dd6336a,
0xb60c4030, 0x7bb552f1, 0xf1b91481, 0xc8929b82,
0xa1c22bf8, 0xd585aae4, 0x17fb4f6a, 0xa9c0d32e,
0x2036f9b5, 0x3a95d611, 0xd554f25d, 0x3441153f,
0x7fa89ff4, 0x8f91241c, 0x4b2cc5a9, 0xd3035a00,
0x5c80707f, 0xe610fd47, 0xf60958c2, 0xb55a6fe7,
0x3e1ba335, 0x2dead082, 0x1a8877e8, 0xd0791aad,
0x9706ee52, 0xfb1dc525, 0x7fa1ba54, 0x6a3e9f81,
0xa85a906a, 0x86ce1b46, 0x3b05833e, 0xc8d8fdcb,
0x44e606c5, 0x8807beb2, 0xe46047d3, 0x85b9f5f8,

```
  0x56ed0cba, 0x3cf4e646, 0x970fd9dd, 0xd0600895,
  0x2d0a5f92, 0x891c4220, 0x017fbfe0, 0x4dde2016,
  0x3a6e421d, 0x7e3f2285, 0xdf7956e3, 0x52fdcf83
  };

/*
 * Hash a string using the modified CRC method.  The basic idea of
 * this function is that the hash value is rotated left by 5 bits and
 * then the next character is exclusive-or'ed in to the hash value.
 * The implementation is complicated by the lack of a rotate operation
 * in C++.
 *
 * This hash function does not work well with table sizes that are a
 * power of two.
 */
unsigned int hashStringCRC(         // Hash a string
    const string&   key)       // Key to be hashed
    {
    unsigned int hashValue = 0;
    for (string::const_iterator i = key.begin();  i !=
key.end();  i++)
     {
     /*
      * The following expression could be done in one line, but it
      * would be really nasty, and a modern compiler ought to
      * generate the same code whether it's one line or several.
      * So we'll break it up to make it easier to read.
      *
      * First, we shift the value left to make room for bits from
      * the new key character.
      */
     unsigned int leftShiftedValue = hashValue <<
CRC_HASH_SHIFT;
     /*
      * Shifting left lost the top bits, so we have to extract and
      * position them separately with a right shift.  If we were
      * writing in assembly, we could do all of this in a single
      * rotate instruction, but C++ doesn't give us access to that
      * machine operation so we have to do it the hard way.
      */
     unsigned int rightShiftedValue =
       hashValue >> (WORD_WIDTH - CRC_HASH_SHIFT);
     /*
      * Put the shifted values together, and then XOR them with the
      * next key character (stepping past it in the process).
      */
     hashValue = (leftShiftedValue | rightShiftedValue) ^
(unsigned)*i;
     }
```

```
    return hashValue;
  }

/*
 * Hash a string by interpreting it as a base-256 number.  Unlike the
 * other hash functions in this file, this function must be passed the
 * table size as an argument and performs the modulo function itself
 * (otherwise it wouldn't be able to generate a correct result).
 *
 * Note that instead of multiplying by 256, we shift left by 8 bits
 * (BYTE_WIDTH).  This is faster on almost all machines, and happens to be
 * a bit easier to write in C++ due to the definition of BYTE_WIDTH.
 */
unsigned int hashStringBase256(
    const string&   key,      // Key to be hashed
    unsigned int    tableSize)     // Size of hash table
(hash modulus)
    {
    unsigned int hashValue = 0;
    for (string::const_iterator i = key.begin();  i !=
key.end();  i++)
     {
     hashValue = (hashValue << BYTE_WIDTH) + (unsigned)*i;
     hashValue %= tableSize;
     }
    return hashValue;
    }

/*
 * Hash a string using an algorithm taken from Aho, Sethi, and Ullman,
 * "Compilers: Principles, Techniques, and Tools," Addison-Wesley,
 * 1985, p. 436.  PJW stands for Peter J. Weinberger, who apparently
 * originally suggested the function.
 *
 * The basic idea of this algorithm is similar to that of the modified
 * CRC algorithm, except that instead of shifting the top bits right
 * so that they line up with the newly emptied bottom bits (a rotate),
 * the top bits are shifted only far enough to line up with the top
 * half of the character just XOR-ed into the hash value.
 */
unsigned int hashPJW(              // Hash a string,
Aho/Sethi/Ullman
    const string&   key)      // Key to be hashed
    {
    unsigned int hashValue = 0;
    for (string::const_iterator i = key.begin();  i !=
key.end();  i++)
     {
```

```
     hashValue = (hashValue << PJW_HASH_SHIFT) +
(unsigned)*i;
     unsigned int rotate_bits = hashValue & PJW_HASH_MASK;
     hashValue ^= rotate_bits | (rotate_bits >>
PJW_HASH_RIGHT_SHIFT);
     }
    return hashValue;
    }

/*
* Hash an integer by multiplication.  This algorithm was suggested by
* Knuth.
*/
unsigned int hashIntegerMultiply(  // Hash an integer by
multiplication
    int              key)      // Key to be hashed
    {
    return (unsigned int)key * (unsigned int)(key + 3);
    }

/*
* Hash a string using the BUZ algorithm.  See the helper function for
* more information on the origins and operation of the underlying
* algorithm.
*
* After the string has been hashed according to the BUZ algorithm,
* the resulting integer is re-hashed twice.  Bob Uzgalis claimed in
* his lecture that this produced better hash values.
*
* The BUZ algorithm is the only string-hashing method in this file
* that generates non-sequential hash values when the keys differ by
* one in the last character (e.g., "foo1' and "foo2").  For this
* reason, BUZ is a good choice if you are using linear probing and
* you expect that many of your keys will differ only in the last
* character (such as variable names).
*/
unsigned int hashStringBUZ (
    const string&   key)      // Key to be hashed
    {
    /*
     * Hash the string using the BUZ method (see hashBUZ).
     */
    unsigned int hashValue = BUZINIT;   // Result
    for (string::const_iterator i = key.begin();  i !=
key.end();  i++)
     {
     unsigned int ch = *i;
     if (hashValue & (1 << (WORD_WIDTH - 1)))
```

```
        hashValue = ((hashValue << 1) | 1) ^
BUZ_Table[ch];
     else
        hashValue = (hashValue << 1) ^ BUZ_Table[ch];
     }

  /*
   * Re-hash the hash value twice in hopes of getting more
   * randomness.  By casting &hashValue into a pointer to unsigned
   * character, we effectively treat it as a 4-character string
   * inside hashBUZ, and re-hash that string to get a new hash value.
   */
     hashValue = hashBUZ((unsigned char*)&hashValue, sizeof
hashValue);
    return hashBUZ((unsigned char*)&hashValue, sizeof
hashValue);
    }

/*
 * Hash an integer using the BUZ algorithm.  There's nothing fancy
 * here; see the comments at the end of hashStringBUZ and the comments
 * for hashBuz for more information.  We hash three times in hopes of
 * getting more randomness.
 */
unsigned int hashIntegerBUZ (
    int             key)      /* Key to be hashed */
    {
    unsigned int hashValue = hashBUZ((unsigned char*)&key,
sizeof key);
    hashValue = hashBUZ((unsigned char*)&hashValue, sizeof
hashValue);
    return hashBUZ((unsigned char*)&hashValue, sizeof
hashValue);
    }

/*
 * Do one pass of the BUZ algorithm on a fixed key.
 *
 * The BUZ algorithm was invented by Robert (Bob) Uzgalis.  I don't
 * have a reference for it yet, because I learned it from a lecture,
 * not a paper.  The basic idea is to use a (constant) lookup table of
 * random numbers to generate wider random bits for each input
 * character.
 *
 * The innards of the function are very similar to the modified CRC
 * method, with one minor and one major change.  The minor change is
 * that the hash value is rotated only one bit at a time, rather than
 * 5.  The major change is that instead of XOR-ing the key character
 * in directly, the character is used as an index to look up a 32-bit
```

```
 * random number in a table.  The idea is that you get 32 bits of
 * information into the hash value at each step, rather than 8.  For
 * that reason, hashBUZ should work better with very short string keys
 * and very small integer keys.
 */
static unsigned int hashBUZ (
    const unsigned char* key,       /* Key to be hashed */
    int              keylen)        /* Length of the key */
    {
    unsigned int hashValue = BUZINIT;
    while (--keylen >= 0)
     {
     unsigned int ch = *key++;
     if (hashValue & (1 << (WORD_WIDTH - 1)))
         hashValue = ((hashValue << 1) | 1) ^
BUZ_Table[ch];
     else
         hashValue = (hashValue << 1) ^ BUZ_Table[ch];
     }
    return hashValue;
    }

/*
 * Hash an integer by taking the absolute value.  This is recommended
 * in Bruno Preiss, "Data Structures and Algorithms with
 * Object-Oriented Design Patterns in C++", Wiley, 1999, p. 210.
 *
 * I (Prof. Kuenning) do not have any particular reason to believe
 * that this is a good hash function.
 */
unsigned int hashIntegerAbs(
  int    key)            /* Key to be hashed */
  {
  if (key >= 0)
        return (unsigned int)key;
  else
        return (unsigned int)(-key);
  }

/*
 * Hash a string using the method given in Bruno Preiss, "Data
 * Structures and Algorithms with Object-Oriented Design Patterns in
 * C++", Wiley, 1999, p. 213.
 *
 * This is NOT a very good hash function, especially with non-prime
 * table sizes.
 *
 * This function is somewhat related to the modified CRC function.
 * However, instead of right-shifting the top bits back down so that
```

```
 * they have a further effect on the low bits, they are simply folded
 * back into the same (top) bits of the new hash value.  The net
 * result is that the first few characters of the key will only have
 * an effect on the top bits of the hash value.  If the table size is
 * a power of 2, this important early-character information will be
 * completely lost when the hash value is taken modulo the table size.
 */
unsigned int hashStringPreiss(
   const string&         key)              // Key to be hashed
   {
   unsigned int hashValue = 0;
   for (string::const_iterator i = key.begin();  i != key.end();  i++)
         hashValue = (hashValue & PREISS_HASH_MASK)
          ^ (hashValue << PREISS_HASH_SHIFT)
          ^ (unsigned)*i;
   return hashValue;
   }

/*
 * Hash a string using the method given in Mark Allen Weiss,
 * "Algorithms, Data Structures, and Problem Solving with C++" (first
 * edition), Addison-Wesley, 1996, p. 611.
 *
 * I (Prof. Kuenning) am suspicious of this hash function, but some
 * very quick tests suggest that it may work better than expected.
 *
 * Again, this function is related to the modified CRC function.
 * However, there is no attempt to preserve the top bits of the hash
 * value.  Instead, the bottom few bits of the function are just the
 * XOR of all the input characters.  This means that early information in the
 * key will be almost completely lost.
 */
unsigned int hashStringWeiss1(
   const string&         key)              // Key to be hashed
   {
   unsigned int hashValue = 0;
   for (string::const_iterator i = key.begin();  i != key.end();  i++)
         hashValue = hashValue ^ (hashValue << WEISS_HASH_SHIFT) ^ (unsigned)*i;
   return hashValue;
   }

/*
 * Hash a string using the method given in Mark Allen Weiss,
 * "Algorithms, Data Structures, and Problem Solving with C++" (second
 * edition), Addison-Wesley, 1999, p. 728.
 *
 * I (Prof. Kuenning) am very suspicious of this hash function.
 *
```

```
 * Again, this function is related to the modified CRC function.
 * However, as with hashStringWeiss1, there is no attempt to preserve
 * the top bits of the hash value.  Since the multiplier is not a
 * power of two, the bottom few bits (about 5) are a bit more
 * complicated than the XOR of all the input keys.  But again, the
 * early information in a long key will be lost.
 */
unsigned int hashStringWeiss2(
   const string&          key)                // Key to be hashed
   {
   unsigned int hashValue = 0;
   for (string::const_iterator i = key.begin();  i != key.end();  i++)
         hashValue = hashValue * WEISS_HASH_MULTIPLIER + (unsigned)*i;
   return hashValue;
   }
```