1. Introduction. This simple class represents positions of a 3x3x3 Rubik's cube efficiently (both in space and time). It is the basis of almost every program I have written for the cube over the past few years.

2. Preface of header file. We use a common technique to protect against multiple inclusion and thus redeclaration, by starting the main include file with a conditional that checks whether the file has already been included. We then include the standard headers that we need. We also flatten the namespace so vector and other components are in our namespace for convenience.

```
⟨ cubepos.h 2 ⟩ ≡
#ifndef CUBEPOS_H
#define CUBEPOS_H
#include <cstring>
#include <cstdlib>
#include <stddef.h>
#include <vector>
#include <algorithm>
#include <unistd.h>
#include <sys/time.h>
  using namespace std;
```

See also sections 4 and 5.

3. Distance metric. Early in computer cubing history, two primary move metrics were defined. The *half-turn metric* counts every quarter turn and every half turn of a face as a single move; this was adopted by most west-coast researchers. The *quarter-turn metric* counts a half turn as two moves; this was adopted by east-coast researchers. The quarter-turn metric has some nice properties; for instance, odd positions are exactly those reachable by an odd number of moves. Nonetheless, the half-turn metric is the more common metric, corresponding more closely to what most people think of as a "move".

This class supports only the half turn metric at the moment.

4. Common constants. The following constants are used throughout the class and by other programs that use this. The M constant is the number of automorphisms of the cube induced by rotation and reflection; these automorphisms themselves form a group, commonly called M.

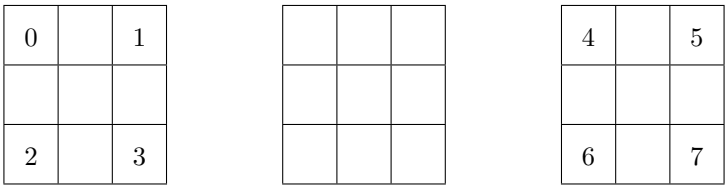
```
⟨ cubepos.h 2 ⟩ +≡
  const int NMOVES = 18;
  const int TWISTS = 3;
  const int FACES = 6;
  const int M = 48;
  const int CUBIES = 24;
```

5. Class declaration. Before the class, we declare a public shared copy of the solved cube (the identity of the group); for convenience, we do not make it a static member object but instead make it a global. Then we define the class. We break it into four components for convenience. Note that we keep the data representation public; we trust users of this class not to abuse this privilege. Many things are simplified by direct access to the data. We also provide a slot for general utility functions, like error reporting, time reporting, and random number generation.

```
⟨ cubepos.h 2 ⟩ +≡
  extern const class cubepos identity_cube;
  ⟨ Global utility declarations 30 ⟩
  class cubepos {
  public:
    ⟨ Public method declarations of cubepos 9 ⟩
    ⟨ Static data declarations of cubepos 12 ⟩
    ⟨ Data representation of cubepos 7 ⟩
  };
  ⟨ Static initialization hack 16 ⟩
#endif
```

6. Representation and numbering. The Rubik's cube has twenty movable cubies that move around the six face centers. If we do not consider rotations of the whole cube, those twenty cubies are the only pieces that move. Whole-cube rotations are not generally considered "moves" so we will ignore these for the moment.

7. Representing the corners. We represent the corners and edges separately. Let us start with the corners. We number the corners 0 to 7 starting with the top layer; the back left corner is 0, then the back right corner is 1, then the front left corner is 2, and the front right corner is 3. We number the corners on the bottom layer in the same order, assigning them 4, 5, 6, and 7.



Top Layer Middle Layer Bottom Layer

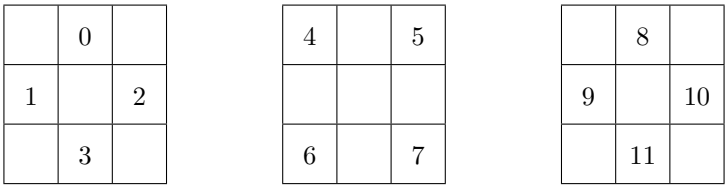
Each cubie can be in one of the eight slots, and it can have up to three distinct orientations, for a total of 24 distinct states (this is `CUBIES`). We therefore use an **unsigned char** to hold these values. The low three bits of the value in $c[i]$ indicates the slot where corner cube i is. The next two bits give the corner twist of that cubie; 0 indicates no twist, 1 indicates a clockwise twist, and 2 indicates a counterclockwise twist. We will define our corner twist convention later.

⟨ Data representation of cubepos 7 ⟩ ≡
unsigned char c[8];

See also section 8.

This code is used in section 5.

8. Representing the edge cubies. The edges are represented in a similar fashion to the corners. There are twelve edges.



Top Layer Middle Layer Bottom Layer

Each edge can be in one of twelve slots, and it can have only two orientations, for a total again of 24 possible states. The low bit of the value in $e[i]$ indicates whether edge i is flipped or not; a 0 indicates it is not, and a 1 indicates it is. The next four bits indicate the edge slot that edge i currently resides in. We will define our edge flip convention later.

⟨ Data representation of cubepos 7 ⟩ +≡
unsigned char e[12];

9. We need methods to compare two **cubepos** objects for equality and for total ordering. We use *memcmp* to do the work; the *g++* compiler (at least) uses efficient intrinsics for this.

⟨ Public method declarations of cubepos 9 ⟩ ≡

```
inline bool operator < (const cubepos &cp) const
{
  return memcmp(this, &cp, sizeof (cp)) < 0;
}
inline bool operator≡(const cubepos &cp) const
{
  return memcmp(this, &cp, sizeof (cp)) ≡ 0;
}
inline bool operator≠(const cubepos &cp) const
{
  return memcmp(this, &cp, sizeof (cp)) ≠ 0;
}
```

See also sections 10, 15, 20, 31, 37, 41, 43, 52, 63, 67, and 74.

This code is used in section 5.

10. Convenience methods and arrays. We provide these four inline functions to make it convenient to go to and from separate permutation/orientation to combined cubie values, and from cubie values to permutations and orientations. We also provide fast routines that lets us add the orientation (only) from one cubieval to another cubieval (for both corners and edges). Finally, we throw in the initialization declaration.

⟨ Public method declarations of cubepos 9 ⟩ +≡

```
static inline int edge_perm(int cubieval)
{
  return cubieval ≫ 1;
}
static inline int edge_ori(int cubieval)
{
  return cubieval & 1;
}
static inline int corner_perm(int cubieval)
{
  return cubieval & 7;
}
static inline int corner_ori(int cubieval)
{
  return cubieval ≫ 3;
}
static inline int edge_flip(int cubieval)
{
  return cubieval ⊕ 1;
}
static inline int edge_val(int perm, int ori)
{
  return perm ∗ 2 + ori;
}
static inline int corner_val(int perm, int ori)
{
  return ori ∗ 8 + perm;
}
static inline int edge_ori_add(int cv1, int cv2)
{
  return cv1 ⊕ edge_ori(cv2);
}
static inline int corner_ori_add(int cv1, int cv2)
{
  return mod24[cv1 + (cv2 & #18)];
}
static inline int corner_ori_sub(int cv1, int cv2)
{
  return cv1 + corner_ori_neg_strip[cv2];
}
static void init();
```

11. We need to put the static data into the C++ file; it's rather annoying that we need to both declare and define these arrays. We also provide an initialization routine that follows the declarations and any other utility functions we need.

```
⟨ cubepos.cpp 11 ⟩ ≡
#include <iostream>
#include "cubepos.h"
#include <math.h>
  ⟨ Static data instantiations 13 ⟩
  ⟨ Local routines for cubepos 36 ⟩
  void cubepos::init( )
  {
    static int initialized = 0;
    if (initialized) return;
    initialized = 1;
    ⟨ Initialization of cubepos 14 ⟩
  }
```

See also sections 17, 23, 35, 38, 39, 40, 42, 45, 46, 54, 65, 66, 68, 69, and 76.

12. Corner orientation changes. Frequently we need to change the orientation of a cubie. This is easy to do for edges (just flip the low-order bit) but for corners it is slightly more difficult. We introduce the following arrays to allow changing corner orientations without performing a modulo or division operation.

```
⟨ Static data declarations of cubepos 12 ⟩ ≡
  static unsigned char corner_ori_inc[CUBIES], corner_ori_dec[CUBIES], corner_ori_neg_strip[CUBIES],
      mod24[2 * CUBIES];
```

See also sections 18, 21, 32, 55, and 71.

This code is used in section 5.

13. We need to declare these in the C++ file. We also declare our identity cube here.

```
⟨ Static data instantiations 13 ⟩ ≡
  const cubepos identity_cube(0, 0, 0);
  unsigned char cubepos::corner_ori_inc[CUBIES], cubepos::corner_ori_dec[CUBIES],
      cubepos::corner_ori_neg_strip[CUBIES], cubepos::mod24[2 * CUBIES];
```

See also sections 19, 22, 24, 25, 26, 27, 33, 44, 47, 48, 49, 56, 57, and 72.

This code is used in section 11.

14. Initialization of these is straightforward.

```
⟨ Initialization of cubepos 14 ⟩ ≡
  for (int i = 0; i < CUBIES; i++) {
    int perm = corner_perm(i);
    int ori = corner_ori(i);
    corner_ori_inc[i] = corner_val(perm, (ori + 1) % 3);
    corner_ori_dec[i] = corner_val(perm, (ori + 2) % 3);
    corner_ori_neg_strip[i] = corner_val(0, (3 - ori) % 3);
    mod24[i] = mod24[i + CUBIES] = i;
  }
```

See also sections 28, 29, 34, 51, 60, 61, 62, and 73.

This code is used in section 11.

15. The constructor. There are two constructors. The first one is both the default constructor and the copy constructor; if no argument is provided, it just makes a copy of the identity cube. The second constructor (with three int arguments all of which are ignored) initializes the current cube more slowly (it does not depend on the identity cube being initialized) and, if the full class itself has not been initialized, initializes the class as well. This second constructor is only used for static objects.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  inline cubepos(const cubepos &cp = identity_cube)
  {
    *this = cp;
  }
  cubepos(int, int, int);
```

16. The static initialization hack. We declare an instance of **cubepos** using this second constructor here, so it appears in every compilation unit that uses **cubepos**, and thus we work around the *static initialization fiasco* that makes C++ so dangerous. We also declare the method that will do the work of initializing all the static members of the class.

```
⟨ Static initialization hack 16 ⟩ ≡
  static cubepos cubepos_initialization_hack(1, 2, 3);
```

This code is used in section 5.

17. Initializing the identity cube. Based on the data structure definitions we have given so far, we can write the constructor that initializes the identity position here. We also throw in a call to the main class initialization routine, which ensures the class is initialized for any compilation unit that includes this header.

```
⟨ cubepos.cpp 11 ⟩ +≡
  cubepos::cubepos(int, int, int)
  {
    for (int i = 0; i < 8; i++) c[i] = corner_val(i, 0);
    for (int i = 0; i < 12; i++) e[i] = edge_val(i, 0);
    init( );
  }
```

18. Face numbering. When we talk about faces of the cube, normally we refer to the color of the face, which, on the 3x3x3 cube, is given by the center cubie, since the center cubie cannot be displaced by normal moves. Rather than adopt a particular color scheme, it is conventional in mathematics about the cube to simply label each face by its spatial location; thus, we have the Up and Down faces, the Left and Right faces, and the Front and Back faces. These are normally abbreviated U, F, R, D, B, and L. We number the faces so that we can use face indices to index into arrays, and so we can number the moves which are all twists of faces. The numbering we choose is arbitrary, but if we choose it to have certain properties it can simplify later code. The numbering we adopt is based on the most regular "unfolding" of the cube we could find. The order we adopt is U, F, R, D, B, L, to which we assign the ordinals 0 to 5. This ordering has the following properties:

1. The opposite face of face i is $i + 3 \, mod$ `FACES`
2. Every three cyclically consecutive faces in this ordering join at a single corner. This defines six of the eight corners; the other two are defined by the odd-numbered and the even-numbered faces, respectively.
3. Every pair of faces whose ordinals do not differ by 3 (mod `FACES`) defines an edge.

```
⟨ Static data declarations of cubepos 12 ⟩ +≡
  static char faces[FACES];
```

19. We initialize the face array here.

```
⟨ Static data instantiations 13 ⟩ +≡
  char cubepos::faces[FACES] = {'U', 'F', 'R', 'D', 'B', 'L'};
```

20. Move numbering. Once we've numbered the faces, numbering the moves is straightforward. We order the twists in the order (clockwise, half turn, counterclockwise) based on the idea that a clockwise twist is the basic move, and a half turn is that move squared (or done twice) and a counterclockwise turn is that move done three times.

We will represent a clockwise turn of the U face by the move U1 (or, where there is no ambiguity, just U). A half-turn is represented by U2, and a counterclockwise turn by U3. The normal convention for a counterclockwise turn is U', but I prefer U3 for simplicity. When discussing these moves in a mathematical context we may use U, U^2, and U^{-1}, respectively.

At this point many programmers would fill the namespace with symbolic names for each of the moves. At this point, I don't believe that carries any benefit.

The move routine has this signature:

⟨ Public method declarations of cubepos 9 ⟩ +≡
void *move*(**int** *mov*);

21. The first move routine. We are ready now to write our first move routine. Since we are storing the location and orientation for each cubie, it is straightforward to determine how a particular move will affect each cubie. We can encode this information into a pair of small arrays, one for corners and one for edges. Note that the impact of a move does not depend on which corner or which edge it is, just the current location and orientation, so we can use the same array for all cubies.

```
⟨ Static data declarations of cubepos 12 ⟩ +≡
  static unsigned char edge_trans[NMOVES][CUBIES], corner_trans[NMOVES][CUBIES];
```

22. Here is the data itself. The two arrays sum to only 864 bytes, and the values for each move are contiguous (within each array) so this is cache-friendly.

```
⟨ Static data instantiations 13 ⟩ +≡
  unsigned char cubepos::edge_trans[NMOVES][CUBIES], cubepos::corner_trans[NMOVES][CUBIES];
```

23. Performing the move. Performing the move itself is simple; we just apply the arrays above to each cubie. We grab a pointer to the array more as a coding shorthand than as a hint to the compiler. We manually unroll the loop; we want to ensure the compiler gives us one short quick function body with no branches or loops; this routine could be called many trillions of times. Even though this code looks relatively long, it's branch-free, cache-friendly, and appears to execute extremely fast on modern processors such as the Intel i7-920.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::move(int mov)
  {
    const unsigned char *p = corner_trans[mov];
    c[0] = p[c[0]];
    c[1] = p[c[1]];
    c[2] = p[c[2]];
    c[3] = p[c[3]];
    c[4] = p[c[4]];
    c[5] = p[c[5]];
    c[6] = p[c[6]];
    c[7] = p[c[7]];
    p = edge_trans[mov];
    e[0] = p[e[0]];
    e[1] = p[e[1]];
    e[2] = p[e[2]];
    e[3] = p[e[3]];
    e[4] = p[e[4]];
    e[5] = p[e[5]];
    e[6] = p[e[6]];
    e[7] = p[e[7]];
    e[8] = p[e[8]];
    e[9] = p[e[9]];
    e[10] = p[e[10]];
    e[11] = p[e[11]];
  }
```

24. Edge permutation. We need to now fill in the *edge_trans* and *corner_trans* arrays. Based on our cubie numbering, we can build an array listing the slots that are affected by a clockwise twist of each face, in the order of the moves, based on our slot numbering convention and move numbering convention. A clockwise twist of the first face (U) moves the cubie from slot 0 into slot 2, from slot 2 into slot 3, from slot 3 into slot 1, and from slot 1 into slot 0. This is represented by the permutation written as $(0, 2, 3, 1)$, and this comprises the first element of the following array. The rest are filled in similarly.

⟨ Static data instantiations 13 ⟩ +≡

static const unsigned char *edge_twist_perm*[FACES][4] = {{0, 2, 3, 1}, {3, 7, 11, 6}, {2, 5, 10, 7}, {9, 11, 10, 8}, {0, 4, 8, 5}, {1, 6, 9, 4}};

25. Corner permutation. We can do the same thing for the corner permutation. A quarter twist of the U face moves the corner in slot 0 to slot 1, from slot 1 to slot 3, from slot 3 to slot 2, and from slot 2 to slot 0. This permutation is $(0, 1, 3, 2)$, and it's the first entry in the array below. This array is carefully constructed so the first two slots are always from the U face (assuming any slots are), which simplifies some later code.

⟨ Static data instantiations 13 ⟩ +≡

static const unsigned char *corner_twist_perm*[FACES][4] = {{0, 1, 3, 2}, {2, 3, 7, 6}, {3, 1, 5, 7}, {4, 6, 7, 5}, {1, 0, 4, 5}, {0, 2, 6, 4}};

26. Edge orientation convention. Now we consider the orientation aspects of moves. When we say a corner is twisted, or an edge is flipped, that makes sense when the cubie is in its solved position. But what does it mean for a cubie to be twisted or flipped when it is in some other slot?

Let us start by considering edge flip. Consider the edge cubie whose home location is the intersection of the U and F faces (we can call this cubie UF). If we permit only the moves U, F, D, and B (and half-twists and counterclockwise twists), it is straightforward to see that whenever the cubie UF is in the U or D face, its U facelet (the sticker colored the same as the center cubie on the U face) is always on the U or D face, and never on one of the F, R, B, or L faces. Further, when the UF cubie is in the middle layer, its U facelet is always on the L or R face. In other words, there is only a single orientation for each cubie in each slot if we start from the solved position and perform only combinations of the moves U, F, D, and B.

If we further permit R and L moves, however, this is no longer true. In particular, the move sequence F1R1U1 brings the UF cubie back to the UF slot, but now the U facelet is in the front face.

We can thus define an edge orientation convention as follows. Only the four moves R1, R3, L1, and L3 modify the edge orientation of any cubie as the cubie moves along slots. All other moves preserve the edge orientation.

There are a number of alternative edge orientation conventions. We can use any pair of opposite faces instead of R and L above. Or, interestingly, we can simply state that every quarter move flips the edge orientation of every involved edge cubie. This last convention has more symmetry than the one we adopt, but for reasons that come from certain specific use of this class, we reject these alternative orientation conventions and use the one defined here.

⟨ Static data instantiations 13 ⟩ +≡

static const unsigned char *edge_change*[FACES] = {0, 0, 1, 0, 0, 1};

27. Corner orientation convention.

Corner orientation is similar, but there are three possible orientations for every cubie, not just two. Note that every cubie has a U or D facelet; this permits a straightforward orientation convention based on simple examination. If the U or D facelet is in the U or D face, we declare the cubie to be properly oriented (an orientation of 0). If twisting the cubie (when looking towards the center of the cube from that cubie) counterclockwise brings it into the oriented state, then we consider the cubie to be oriented clockwise, or +1. If twisting the cubie clockwise brings it into the oriented state, we consider the cubie to be oriented counterclockwise, or +2 (which is $--1 \, mod \, 3$).

From this definition, it is clear that no move of the U or D faces will change the orientation of any corner cubie. A quarter twist of any other face that leaves a particular corner cubie in the same U or D face that it started from will effect a clockwise twist on that cubie. A quarter twist that moves a corner cube from the U face to the D face, or from the D face to the U face, will effect a counterclockwise twist on that cubie. This can be summarized in the following array. Note that we use the information that the *corner_twist_perm* array above always starts with two U face slots before listing two D face slots; thus, the transition corresponding to elements 0 and 2 preserve the U or D face of a cubie, while the elements for 1 and 3 move a cubie from the U face to the D face or vice versa.

⟨ Static data instantiations 13 ⟩ +≡

```
static const unsigned char corner_change[FACES][4] = {{0,0,0,0},{1,2,1,2},{1,2,1,2},{0,0,0,0},
    {1,2,1,2},{1,2,1,2},};
```

28. Making the move table. At this point we have all the data we need to fill in the *corner_trans* and *edge_trans* arrays. This initialization routine combines information from the four static arrays above, along with the bit assignment for the cubie information we have defined. First we fill in default unchanged values for all entries.

⟨ Initialization of cubepos 14 ⟩ +≡

```
for (int m = 0; m < NMOVES; m++)
  for (int c = 0; c < CUBIES; c++) {
    edge_trans[m][c] = c;
    corner_trans[m][c] = c;
  }
```

29. Next we modify the ones affected by the twists, according to the *corner_twist_perm* and *edge_twist_perm* arrays. For every move, we figure out if it is a quarter move or not, and calculate the increment in the permutation arrays (1 for clockwise, 2 for half turns, and 3 for counterclockwise). For both corners and edges, we only change the orientation if it is a quarter move, and if so, according to the *edge_change* and *corner_change* arrays, respectively.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  for (int f = 0; f < FACES; f++)
    for (int t = 0; t < 3; t++) {
      int m = f * TWISTS + t;
      int isquarter = (t ≡ 0 ∨ t ≡ 2);
      int perminc = t + 1;
      if (m < 0) continue;
      for (int i = 0; i < 4; i++) {
        int ii = (i + perminc) % 4;
        for (int o = 0; o < 2; o++) {
          int oo = o;      /* new orientation */
          if (isquarter) oo ⊕= edge_change[f];
          edge_trans[m][edge_val(edge_twist_perm[f][i], o)] = edge_val(edge_twist_perm[f][ii], oo);
        }
        for (int o = 0; o < 3; o++) {
          int oo = o;      /* new orientation */
          if (isquarter) oo = (corner_change[f][i] + oo) % 3;
          corner_trans[m][corner_val(corner_twist_perm[f][i], o)] = corner_val(corner_twist_perm[f][ii], oo);
        }
      }
    }
```

30. Inverse positions. What we have so far makes a useful and powerful class. But there are a lot of interesting and useful operations we can provide that reflect the group nature of Rubik's cube. In particular, the inverse operation is very important.

Every move sequence leads to a particular position from the solved cube. Just like each move has an inverse move, each position has an inverse position and each sequence has an inverse sequence.

For moves, we declare, instantiate, and initialize the inverse move. Halfturn moves are their own inverse; the inverse of a clockwise move is a counterclockwise move, and vice versa.

We use a typedef for a movesequence to make the code clearer and to help eliminate the template syntax that trips up cweave.

```
⟨ Global utility declarations 30 ⟩ ≡
  typedef vector⟨int⟩ moveseq;
```

See also sections 64, 70, and 75.

This code is used in section 5.

31. First we declare the public methods to invert a move, move sequence, and position. Since inverting a position will be so common, we require that the destination be passed in by reference (which eliminates any confusion on the part of the programmer as to when an object will be created or destroyed).

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  static int invert_move(int mv)
  {
    return inv_move[mv];
  }
  static moveseq invert_sequence(const moveseq &sequence);
  void invert_into(cubepos &dst) const;
```

32. To invert moves, we need a quick static array.

```
⟨ Static data declarations of cubepos 12 ⟩ +≡
  static unsigned char inv_move[NMOVES];
```

33. We add instantiate this array.

```
⟨ Static data instantiations 13 ⟩ +≡
  unsigned char cubepos::inv_move[NMOVES];
```

34. Initialization of this array is straightforward. We simply negate the twist component.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  for (int i = 0; i < NMOVES; i++) inv_move[i] = TWISTS * (i/TWISTS) + (NMOVES − i − 1) % TWISTS;
```

35. To invert a sequence, we reverse it and invert each move. This routine returns a new vector; we do not anticipate it being called frequently since it is usually more convenient to just invert a position.

```
⟨ cubepos.cpp 11 ⟩ +≡
  moveseq cubepos::invert_sequence(const moveseq &seq)
  {
    unsigned int len = seq.size();
    moveseq r(len);
    for (unsigned int i = 0; i < len; i++) r[len − i − 1] = invert_move(seq[i]);
    return r;
  }
```

36. Now we get into one of the more entertaining routines: inverting a position. If we ignore orientation, then the cube is just a combination of two permutations. Inverting a permutation stored in array a to an array b is easy; just set $b[a[i]] = i$ for $i in 0..n-1$. Let us consider what happens to orientations.

Let us consider edges first. Just like permutations can be composed, cube positions in Rubik's cube can be composed (indeed, this is much of what it means for the cube to be a group). Thus, when we speak of a position that has cubie number 3 at slot 5, we can also treat it as an operation that moves the cubie that was at slot 3 to slot 5, just as we can for permutations. Consider the permutation $(0, 3, 1)$, which means the cubie at position 0 is moved to position 3, the cubie at position 3 is moved to position 1, and the cubie at position 1 is moved to position 0. The inverse permutation is clearly $(0, 1, 3)$. Let us extend the permutation notation with $+$ to indicate an edge flip or corner twist, and $-$ to indicate a counterclockwise corner twist. Thus the position (looking only at edges for the moment) $(0, 3+, 1)$ means that the cubie at slot 0 is moved to slot 3 and flipped, the cubie at slot 3 is moved to slot 1, and the cubie at slot 1 is moved to slot 0.

With groups, one way to calculate the inverse is to repeately calculate powers of an element until you get back to the identity element; the position immediately preceding the identity element in this sequence is the inverse of the original position. Let us do this with the $(0, 3+, 1)$ position. Applying it to itself yields $(0, 1+, 3+)$. Applying it again gives $(0+), (1+), (3+)$ (that is, all cubies are in their original positions but flipped). Continuing, we obtain $(0+, 3, 1+)$, $(0+, 1, 3)$, and finally 1 (that is, the identity permutation). Thus, the order of this position is 6, and the inverse of this position is $(0+, 1, 3)$.

We can also perform this inversion by simple inspection. If a position takes a cubie from slot i to slot j and flips it in the process, then the inverse position must take the cubie in slot j and move it to slot i, unflipping it in the process.

This allows us to write our inversion routine very easily.

⟨ Local routines for cubepos 36 ⟩ ≡

```
void cubepos::invert_into(cubepos &dst) const
{
  for (int i = 0; i < 8; i++) {
    int cval = c[i];
    dst.c[corner_perm(cval)] = corner_ori_sub(i, cval);
  }
  for (int i = 0; i < 12; i++) {
    int cval = e[i];
    dst.e[edge_perm(cval)] = edge_val(i, edge_ori(cval));
  }
}
```

See also sections 50, 53, 58, and 59.

This code is used in section 11.

37. The second move routine. Now, a move sequence applied to the identity cube yields a position, and applying the inverse sequence yields the inverse position. We can think of a position as corresponding to an implicit move sequence that leads to that position, and performing a move on the position extends that sequence on the right by that move. What about extending the sequence on the left; can we perform that operation?

Alternatively, our class as we have presented it so far contains two arrays c and e that, for each corner and edge cubie, indicates what slot it is in and what orientation it has in that slot. Another representation might instead use those arrays to indicate for each slot, what cubie is in the slot and what orientation it has. Interestingly, these two distinct representations are exactly inverses of each other. Using the *invert_into* method, we can convert a position from one representation to the other.

Can we write a move routine that treats the position in the second fashion? Yes we can, and it is fairly simple to do. We will call this routine *movepc* because it is a move routine that treats the representation as a slot (position) to cubie map (where our default presentation so far has considered it as a cubie to slot map).

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  void movepc(int mov);
```

38. One advantage of representing the cube with a slot to cubie map is, for each move, we only need to update the specific eight slots that are modified by a particular move. In our cubie to slot representation, we cannot easily take such a shortcut. This advantage is reduced by the need to branch based on the move number itself, however; which slots are affected by a move depends on the move. It is possible to use a table and indirection to eliminate the switch, but half moves and quarter moves still must be distinguished in some way, so you can't eliminate it all.

Moves on the up and down face preserve both corner and edge orientations of the cubies, so they just move cubie values from slot to slot. We define macros to do a swap and a four-cycle. The first argument to these macros is the array to modify; the remaining arguments are indices into this array.

```
⟨ cubepos.cpp 11 ⟩ +≡
#define ROT2(cc, a, b)
  {
    unsigned char t = cc[a];
    cc[a] = cc[b];
    cc[b] = t;
  }
#define ROT4(cc, a, b, c, d)
  {
    unsigned char t = cc[d];
    cc[d] = cc[c];
    cc[c] = cc[b];
    cc[b] = cc[a];
    cc[a] = t;
  }
#define ROT22(cc, a, b, c, d)ROT2(cc, a, c)ROT2 (cc, b, d)
```

39. Some moves change the orientation of edges. Some moves change the orientation of corners. Looking at the definition of the *corner_change* array, we note that the moves that change the orientations of the corners are all the same; the even corners are incremented and the odd ones are decremented, so we only need a single preprocessor macro.

```
⟨ cubepos.cpp 11 ⟩ +≡
#define EDGE4FLIP(a, b, c, d)
  {
    unsigned char t = e[d];
    e[d] = edge_flip(e[c]);
    e[c] = edge_flip(e[b]);
    e[b] = edge_flip(e[a]);
    e[a] = edge_flip(t);
  }
#define CORNER4FLIP(a, b, cc, d)
  {
    unsigned char t = c[d];
    c[d] = corner_ori_inc[c[cc]];
    c[cc] = corner_ori_dec[c[b]];
    c[b] = corner_ori_inc[c[a]];
    c[a] = corner_ori_dec[t];
  }
```

40. With these macros, we are ready to write the routine; just a big switch statement, then a bunch of constants indicating which slots are affected. We do need to separate the halfturn from the quarter turn case. All of these constants are tedious to look up so it's imperative we test them exhaustively; luckily, the redundancy from the *move* routine and the relationship between *move* and *movepc* makes it easy to test. Note how the slice turn moves are just combinations of the other moves.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::movepc(int mov)
  {
    switch (mov) {
    case 0: ROT4(e, 0, 2, 3, 1);
      ROT4(c, 0, 1, 3, 2);
      break;
    case 1: ROT22(e, 0, 2, 3, 1);
      ROT22(c, 0, 1, 3, 2);
      break;
    case 2: ROT4(e, 1, 3, 2, 0);
      ROT4(c, 2, 3, 1, 0);
      break;
    case 3: ROT4(e, 3, 7, 11, 6);
      CORNER4FLIP(3, 7, 6, 2);
      break;
    case 4: ROT22(e, 3, 7, 11, 6);
      ROT22(c, 2, 3, 7, 6);
      break;
    case 5: ROT4(e, 6, 11, 7, 3);
      CORNER4FLIP(3, 2, 6, 7);
      break;
    case 6: EDGE4FLIP(2, 5, 10, 7);
      CORNER4FLIP(1, 5, 7, 3);
      break;
    case 7: ROT22(e, 2, 5, 10, 7);
      ROT22(c, 3, 1, 5, 7);
      break;
    case 8: EDGE4FLIP(7, 10, 5, 2);
      CORNER4FLIP(1, 3, 7, 5);
      break;
    case 9: ROT4(e, 9, 11, 10, 8);
      ROT4(c, 4, 6, 7, 5);
      break;
    case 10: ROT22(e, 9, 11, 10, 8);
      ROT22(c, 4, 6, 7, 5);
      break;
    case 11: ROT4(e, 8, 10, 11, 9);
      ROT4(c, 5, 7, 6, 4);
      break;
    case 12: ROT4(e, 0, 4, 8, 5);
      CORNER4FLIP(0, 4, 5, 1);
      break;
    case 13: ROT22(e, 0, 4, 8, 5);
      ROT22(c, 1, 0, 4, 5);
      break;
    case 14: ROT4(e, 5, 8, 4, 0);
```

```
      CORNER4FLIP(0, 1, 5, 4);
      break;
    case 15: EDGE4FLIP(1, 6, 9, 4);
      CORNER4FLIP(2, 6, 4, 0);
      break;
    case 16: ROT22(e, 1, 6, 9, 4);
      ROT22(c, 0, 2, 6, 4);
      break;
    case 17: EDGE4FLIP(4, 9, 6, 1);
      CORNER4FLIP(2, 0, 4, 6);
      break;
    }
  }
```

41. The multiplication operation. Rubik's cube is a group, and groups are defined by a multiplication operation between arbitrary elements. So far all we have are element definitions, move definitions, and inversion operations. It is straightforward to write the general multiplication routine. Since this might be performance-critical, we again require that the result be passed in by reference, so the programmer is explicitly aware of allocation and deallocation.

If the only operations performed for an investigation are *move*, creation of an identity cube, equality comparison, and ordering comparisons where the order doesn't matter (but existence of a total order does), then *movepc* can be used instead of *move*; they simply reflect different representations of the same group. The *movepc* routine essentially acts with inverse generators left-multiplied by inverse positions, rather than the normal generators right-multiplied by positions. Essentially, we are operating on an automorphism of the main group. So if we are using *movepc* to represent moves, we need to use an alternative multiplication operation that reverses the operands (to reflect that *movepc* is actually doing left multiplication rather than right multiplication from the perspective of our normal representation).

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  static void mul(const cubepos &a, const cubepos &b, cubepos &r);
  inline static void mulpc(const cubepos &a, const cubepos &b, cubepos &r)
  {
    mul(b, a, r);
  }
```

42. The multiplication routine itself is straightforward; it is the same as a normal permutation multiplication, except we also need to carry forward and "add" the orientations.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::mul(const cubepos &a, const cubepos &b, cubepos &r)
  {
    for (int i = 0; i < 8; i++) {
      int cc = a.c[i];
      r.c[i] = corner_ori_add(b.c[corner_perm(cc)], cc);
    }
    for (int i = 0; i < 12; i++) {
      int cc = a.e[i];
      r.e[i] = edge_ori_add(b.e[edge_perm(cc)], cc);
    }
  }
```

43. Parsing and printing moves and move sequences. Cube programs frequently require input and output of moves and move sequences. This section provides some simple routines to support this.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  static void skip_whitespace(const char *&p);
  static int parse_face(const char *&p);
  static int parse_face(char f);
  static void append_face(char *&p, int f)
  {
    *p++ = faces[f];
  }
  static int parse_move(const char *&p);
  static void append_move(char *&p, int mv);
  static moveseq parse_moveseq(const char *&p);
  static void append_moveseq(char *&p, const moveseq &seq);
  static char *moveseq_string(const moveseq &seq);
```

44. We start with a buffer that's usually big enough to hold a result. Note that use of this static buffer means the *_string*( ) methods are not thread-safe.

```
⟨ Static data instantiations 13 ⟩ +≡
  static char static_buf[200];
```

45. The routines themselves are straightforward.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::skip_whitespace(const char *&p)
  {
    while (*p ∧ *p ≤ ' ') p++;
  }
  int cubepos::parse_face(const char *&p)
  {
    int f = parse_face(*p);
    if (f ≥ 0) p++;
    return f;
  }
  int cubepos::parse_face(char f)
  {
    switch (f) {
    case 'u': case 'U': return 0;
    case 'f': case 'F': return 1;
    case 'r': case 'R': return 2;
    case 'd': case 'D': return 3;
    case 'b': case 'B': return 4;
    case 'l': case 'L': return 5;
    default: return -1;
    }
  }
  int cubepos::parse_move(const char *&p)
  {
    skip_whitespace(p);
    const char *q = p;
    int f = parse_face(q);
    if (f < 0) return -1;
    int t = 0;
    switch (*q) {
    case '1': case '+': t = 0;
      break;
    case '2': t = 1;
      break;
    case '3': case '\'': case '-': t = TWISTS - 1;
      break;
    default: return -1;
    }
    p = q + 1;
    return f * TWISTS + t;
  }
```

46. And they keep going on.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::append_move(char *&p, int mv)
  {
    append_face(p, mv/TWISTS);
    *p++ = "123"[mv % TWISTS];
    *p = 0;
  }
  moveseq cubepos::parse_moveseq(const char *&p)
  {
    moveseq r;
    int mv;
    while ((mv = parse_move(p)) ≥ 0) r.push_back(mv);
    return r;
  }
  void cubepos::append_moveseq(char *&p, const moveseq &seq)
  {
    *p = 0;
    for (unsigned int i = 0; i < seq.size(); i++) append_move(p, seq[i]);
  }
  char *cubepos::moveseq_string(const moveseq &seq)
  {
    if (seq.size() > 65)
      error ("! can't print a move sequence that long") ;
    char *p = static_buf;
    append_moveseq(p, seq);
    return static_buf;
  }
```

47. Singmaster notation. The standard format for cube positions in called Singmaster positional notation after David Singmaster's early work on cube math. His notation represents a solved cube in cubie order, giving the cubie in each slot along with the orientation; he orders the slots in a particular way, and lists a default orientation for each cubie. In his notation, a solved cube is represented by the string

⟨ Static data instantiations 13 ⟩ +≡

```
static const char *sing_solved = "UF␣UR␣UB␣UL␣DF␣DR␣DB␣DL␣FR␣FL␣BR␣BL␣UFR\
    ␣URB␣UBL␣ULF␣DRF␣DFL␣DLB␣DBR";
```

48. The edge cubies are listed first, followed by the corner cubies. Note that his representation does not have any implicit orientation convention; it is a direct permutation mapping of all 48 cubie stickers on the edge and corner cubies. His ordering differs from ours, but we can give our cubie numbering (complete with orientation) in a pair of literal arrays. For the corners, we actually list 48 values; the other 24 will come into play later, when we consider symmetry.

⟨ Static data instantiations 13 ⟩ +≡

```
static const char *const smedges[] = {"UB","BU","UL","LU","UR","RU","UF","FU","LB","BL",
    "RB","BR","LF","FL","RF","FR","DB","BD","DL","LD","DR","RD","DF","FD",};
static const char *const smcorners[] = {"UBL","URB","ULF","UFR","DLB","DBR","DFL","DRF",
    "LUB","BUR","FUL","RUF","BDL","RDB","LDF","FDR","BLU","RBU","LFU","FRU","LBD","BRD",
    "FLD","RFD","ULB","UBR","UFL","URF","DBL","DRB","DLF","DFR","LBU","BRU","FLU","RFU",
    "BLD","RBD","LFD","FRD","BUL","RUB","LUF","FUR","LDB","BDR","FDL","RDF",};
```

49. We can parse the Singmaster notation by parsing cubies. One good way to parse cubies is to turn each cubie into a base-6 number with a leading 1; the leading 1 gives us information on how many face specifications we saw. We also need information on the order of the cubies in the Singmaster notation. The following arrays collect some information useful in this parsing. We use the value 99 as a marker of an invalid cubie.

⟨ Static data instantiations 13 ⟩ +≡

```
const int INVALID = 99;
static unsigned char lookup_edge_cubie[FACES * FACES];
static unsigned char lookup_corner_cubie[FACES * FACES * FACES];
static unsigned char sm_corner_order[8];
static unsigned char sm_edge_order[12];
static unsigned char sm_edge_flipped[12];
```

50. The following routines parse a generic cubie, returning a base-6 number with a leading 1, and then routines that parse edges and cubies, respectively.

```
⟨ Local routines for cubepos 36 ⟩ +≡
  static int parse_cubie(const char *&p)
  {
    cubepos::skip_whitespace(p);
    int v = 1;
    int f = 0;
    while ((f = cubepos::parse_face(p)) ≥ 0) {
      v = v * 6 + f;
      if (v ≥ 2 * 6 * 6 * 6) return -1;
    }
    return v;
  }
  static int parse_edge(const char *&p)
  {
    int c = parse_cubie(p);
    if (c < 6 * 6 ∨ c ≥ 2 * 6 * 6) return -1;
    c = lookup_edge_cubie[c - 6 * 6];
    if (c ≡ INVALID) return -1;
    return c;
  }
  static int parse_corner(const char *&p)
  {
    int c = parse_cubie(p);
    if (c < 6 * 6 * 6 ∨ c ≥ 2 * 6 * 6 * 6) return -1;
    c = lookup_corner_cubie[c - 6 * 6 * 6];
    if (c ≡ INVALID ∨ c ≥ CUBIES) return -1;
    return c;
  }
```

51. We need to initialize all of those arrays.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  memset(lookup_edge_cubie, INVALID, sizeof (lookup_edge_cubie));
  memset(lookup_corner_cubie, INVALID, sizeof (lookup_corner_cubie));
  for (int i = 0; i < CUBIES; i++) {
    const char *tmp = 0;
    lookup_corner_cubie[parse_cubie(tmp = smcorners[i]) - 6 * 6 * 6] = i;
    lookup_corner_cubie[parse_cubie(tmp = smcorners[CUBIES + i]) - 6 * 6 * 6] = CUBIES + i;
    lookup_edge_cubie[parse_cubie(tmp = smedges[i]) - 6 * 6] = i;
  }
  const char *p = sing_solved;
  for (int i = 0; i < 12; i++) {
    int cv = parse_edge(p);
    sm_edge_order[i] = edge_perm(cv);
    sm_edge_flipped[i] = edge_ori(cv);
  }
  for (int i = 0; i < 8; i++) sm_corner_order[i] = corner_perm(parse_corner(p));
```

52. We provide methods to parse a cube position directly from the Singmaster notation, and write a position in Singmaster notation. The return value from the parse method is either 0, indicating success, or a string indicating the error that was encountered.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  const char *parse_Singmaster(const char *p);
  char *Singmaster_string( ) const;
```

53. To parse the Singmaster notation, we simply parse the input, reading each cubie, and store it in the appropriate slot after doing any orientation correction that might be needed. If it leads with `SING` we permit this and skip it. Note that our internal representation is cubie to position, and Singmaster notation shows position to cubie, so we invert as we read.

```
⟨ Local routines for cubepos 36 ⟩ +≡
  const char *cubepos::parse_Singmaster(const char *p)
  {
    if (strncmp(p, "SING␣", 5) ≡ 0) p += 5;
    int m = 0;
    for (int i = 0; i < 12; i++) {
      int c = parse_edge(p) ⊕ sm_edge_flipped[i];
      if (c < 0) return "No␣such␣edge";
      e[edge_perm(c)] = edge_val(sm_edge_order[i], edge_ori(c));
      m |= 1 ≪ i;
    }
    for (int i = 0; i < 8; i++) {
      int cval = parse_corner(p);
      if (cval < 0) return "No␣such␣corner";
      c[corner_perm(cval)] = corner_ori_sub(sm_corner_order[i], cval);
      m |= 1 ≪ (i + 12);
    }
    skip_whitespace(p);
    if (*p) return "Extra␣stuff␣after␣Singmaster␣representation";
    if (m ≠ ((1 ≪ 20) − 1)) return "Missing␣at␣least␣one␣cubie";
    return 0;
  }
```

54. Writing Singmaster notation (or more precisely, returning a static string containing the Singmaster notation formation of a position) is straightforward. We do need to invert at the start so we have the data in the proper order.

```
⟨ cubepos.cpp 11 ⟩ +≡
  char *cubepos::Singmaster_string() const
  {
    cubepos cp;
    invert_into(cp);
    char *p = static_buf;
    for (int i = 0; i < 12; i++) {
      if (i ≠ 0) *p++ = '␣';
      int j = sm_edge_order[i];
      const char *q = smedges[cp.e[j] ⊕ sm_edge_flipped[i]];
      *p++ = *q++;
      *p++ = *q++;
    }
    for (int i = 0; i < 8; i++) {
      *p++ = '␣';
      int j = sm_corner_order[i];
      const char *q = smcorners[cp.c[j]];
      *p++ = *q++;
      *p++ = *q++;
      *p++ = *q++;
    }
    *p = 0;
    return static_buf;
  }
```

55. Symmetry. Our next consideration is whole cube rotations and the symmetries of the cube. Our labeling of the cube uses the names up, down, left, right, back, and front intentionally—we care only about what face is up, not what color it is. Since the color of a face is defined by the color of the center cubie, whole cube rotations change the mappings of colors to spatial orientations.

Since our representation does not denote the actual colors that are on each face, we cannot represent cube rotations directly. We only represent how the cubies move. But we *can* represent conjugations of whole cube rotations—that is, given a cube rotation $m \in M$, we can compute the operation mpm' for any cube position p, and this suffices for us to take advantage of the symmetries of the cube. In this expression, the m operation rotates the whole cube and thus moves the face centers; p performs operations on all cubies except the face centers, and then m' rotates a cube again, moving the face centers back to where they originally were.

The whole cube rotations M are themselves a group, but a slightly larger one than may first appear. If you consider a physical cube and its rotations, you can choose any specific color to be the up face, and once that is done, you have four possibilities for the front face. Thus, we see a physical cube has 24 possible rotations. We only consider rotations that map faces onto faces, not partial rotations. If we examine the cube in a mirror, we see an additional 24 "mirror image" rotations that are possible—these for instance might exchange front for back, but leave left, right, up, and down the same.

The cube has three orthogonal axes—one passing through the up and down faces (which we call UD), one passing through the left and right faces (LR), and one passing through the front and back faces (FB). If we consider the U, F, and R faces to be the positive side of each of these axes, then we can decompose the rotation group M into a map of axes onto themselves, and an indication of which of the axes are inverted.

We assign ordinal numbers to the elements of M in a useful way. We assign the ordinal 0 to the identity element of M. We collect the elements of m that share the same axis mappings into contiguous groups of eight; we have six such groups.

To manage remappings, we have an array that represents the face remappings for each $m \in M$, another that represents the multiplication operation for M, another that represents the inverse operation in M, and another that represents the move mapping for M. Finally, we need mapping arrays for the corners and edges for all the different remappings.

⟨ Static data declarations of cubepos 12 ⟩ +≡
```
static unsigned char face_map[M][FACES], move_map[M][NMOVES];
static unsigned char invm[M], mm[M][M];
static unsigned char rot_edge[M][CUBIES], rot_corner[M][CUBIES];
```

56. These arrays must be instantiated.

⟨ Static data instantiations 13 ⟩ +≡
```
unsigned char cubepos::face_map[M][FACES], cubepos::move_map[M][NMOVES];
unsigned char cubepos::mm[M][M], cubepos::invm[M];
unsigned char cubepos::rot_edge[M][CUBIES], cubepos::rot_corner[M][CUBIES];
```

57. Everything comes from facemap. We cluster the elements of M in groups of eight such that each group has the same axis mapping. The axis mappings are defined by the following six cubies. Note that the first two maintain the up/down axei, and they come in pairs that share the same up/down axis. Within each group, we negate different subsets of the axes. We order these carefully, so the even ones are the "normal" cube rotations, and the odd ones are the "mirrored" cube rotations. Indeed, the *axis_negate_map* uses a gray-code-style ordering.

⟨ Static data instantiations 13 ⟩ +≡
```
static const char *const axis_permute_map[] = {"UFR","URF","FRU","FUR","RUF","RFU"};
static const char *const axis_negate_map[] = {"UFR","UFL","UBL","UBR","DBR","DBL","DFL","DFR"};
```

58. Given the three faces in the UFR corner, we can easily fill out a face map entry. We just parse the corner one face at a time, inserting those values in the face map entry and the opposite face offset by three.

```
⟨ Local routines for cubepos 36 ⟩ +≡
  static void parse_corner_to_facemap(const char *p, unsigned char *a)
  {
    for (int i = 0; i < 3; i++) {
      int f = cubepos::parse_face(p[i]);
      a[i] = f;
      a[i + 3] = (f + 3) % FACES;
    }
  }
```

59. We also provide a routine that does permutation multiplication for a facemap.

```
⟨ Local routines for cubepos 36 ⟩ +≡
  static void face_map_multiply(unsigned char *a, unsigned char *b, unsigned char *c)
  {
    for (int i = 0; i < 6; i++) c[i] = b[a[i]];
  }
```

60. With this code, generating the face map is easy; we simply generate the basic axis permutation elements, then the axis negation elements, and finally we do multiplication to fill out the table. The move map is also easy to compute, since we ordered the elements in such a way that determining which elements are negative (or mirror) mappings is straightforward.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  unsigned char face_to_m[FACES * FACES * FACES];
  for (int i = 0; i < 6; i++) parse_corner_to_facemap(axis_permute_map[i], face_map[8 * i]);
  for (int i = 0; i < 8; i++) parse_corner_to_facemap(axis_negate_map[i], face_map[i]);
  for (int i = 1; i < 6; i++)
    for (int j = 1; j < 8; j++) face_map_multiply(face_map[8 * i], face_map[j], face_map[8 * i + j]);
```

61. Now we calculate the multiplication table and the inverse table for M. We also calculate the move map.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  for (int i = 0; i < M; i++) {
    int v = face_map[i][0] * 36 + face_map[i][1] * 6 + face_map[i][2];
    face_to_m[v] = i;
  }
  unsigned char tfaces[6];
  for (int i = 0; i < M; i++)
    for (int j = 0; j < M; j++) {
      face_map_multiply(face_map[i], face_map[j], tfaces);
      int v = tfaces[0] * 36 + tfaces[1] * 6 + tfaces[2];
      mm[i][j] = face_to_m[v];
      if (mm[i][j] ≡ 0) invm[i] = j;
    }
  for (int m = 0; m < M; m++) {
    int is_neg = (m ⊕ (m ≫ 3)) & 1;
    for (int f = 0; f < 6; f++) {
      for (int t = 0; t < TWISTS; t++) {
        if (is_neg) move_map[m][f * TWISTS + t] = face_map[m][f] * TWISTS + TWISTS - 1 - t;
        else move_map[m][f * TWISTS + t] = face_map[m][f] * TWISTS + t;
      }
    }
  }
```

62. Once we have *face_map* constructed, the actual rotation operations are straightforward. For each cubie, we remap the faces, and then look up the result.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  for (int m = 0; m < M; m++)
    for (int c = 0; c < CUBIES; c++) {
      int v = 0;
      for (int i = 0; i < 2; i++) v = 6 * v + face_map[m][parse_face(smedges[c][i])];
      rot_edge[m][c] = lookup_edge_cubie[v];
      v = 0;
      for (int i = 0; i < 3; i++) v = 6 * v + face_map[m][parse_face(smcorners[c][i])];
      rot_corner[m][c] = mod24[lookup_corner_cubie[v]];
    }
```

63. Now that we have the mapping arrays created, we can write our functions. The *remap_into* function remaps a position p according to mpm', passing in the destination by reference. The *canon_into48* method calculates the canonical (first, or least) **cubepos** for all $m \in M$. The *canon_into96* returns the least of the canonicalization of the cubepos and its inverse; it requires an auxilliary routine *canon_into48_aux* which we also define.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  void remap_into(int m, cubepos &dst) const;
  void canon_into48(cubepos &dst) const;
  void canon_into48_aux(cubepos &dst) const;
  void canon_into96(cubepos &dst) const;
```

64. We declare a pair of constants for move masks.

```
⟨ Global utility declarations 30 ⟩ +≡
  const int ALLMOVEMASK = (1 ≪ NMOVES) − 1;
  const int ALLMOVEMASK_EXT = (1 ≪ NMOVES) − 1;
```

65. The *remap_into* function looks a bit complicated but runs quickly.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::remap_into(int m, cubepos &dst) const
  {
    int mprime = invm[m];
    for (int i = 0; i < 8; i++) {
      int c1 = rot_corner[mprime][i];
      int c2 = corner_ori_add(c[corner_perm(c1)], c1);
      dst.c[i] = rot_corner[m][c2];
    }
    for (int i = 0; i < 12; i++) {
      int c1 = rot_edge[mprime][i * 2];
      int c2 = edge_ori_add(e[edge_perm(c1)], c1);
      dst.e[i] = rot_edge[m][c2];
    }
  }
```

66. To canonicalize 48 ways, we could just call the above routine 48 times, and choose the least. But we can eliminate most of the possibilities more quickly with a slightly more complex routine.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::canon_into48_aux(cubepos &dst) const
  {
    for (int m = 1; m < M; m++) {
      int mprime = invm[m];
      int isless = 0;
      for (int i = 0; i < 8; i++) {
        int c1 = rot_corner[mprime][i];
        int c2 = corner_ori_add(c[corner_perm(c1)], c1);
        int t = rot_corner[m][c2];
        if (isless ∨ t < dst.c[i]) {
          dst.c[i] = t;
          isless = 1;
        }
        else if (t > dst.c[i]) goto nextm;
      }
      for (int i = 0; i < 12; i++) {
        int c1 = rot_edge[mprime][i * 2];
        int c2 = edge_ori_add(e[edge_perm(c1)], c1);
        int t = rot_edge[m][c2];
        if (isless ∨ t < dst.e[i]) {
          dst.e[i] = t;
          isless = 1;
        }
        else if (t > dst.e[i]) goto nextm;
      }
    nextm: ;
    }
  }
  void cubepos::canon_into48(cubepos &dst) const
  {
    dst = *this;
    canon_into48_aux(dst);
  }
```

67. Frequently we need a random cube position. This should be a truly random position, rather than one generated by a finite number of random moves. We make this a destructive *randomize*( ) call on the current cubepos.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  void randomize( );
```

68. To implement this, we shuffle the cubies around, and then update all the orientations randomly. When we shuffle the cubies it's critical we maintain parity. So we only do random selection for the first ten edges; parity forces the eleventh edge, and of course the twelfth is forced because it is the only one left.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::randomize()
  {
    int parity = 0;
    for (int i = 0; i < 7; i++) {
      int j = i + (int)((8 - i) * myrand());
      if (i ≠ j) {
        swap(c[i], c[j]);
        parity++;
      }
    }
    for (int i = 0; i < 11; i++) {
      int j = i + (int)((12 - i) * myrand());
      if (i ≠ j) {
        swap(e[i], e[j]);
        parity++;
      }
    }
    if (parity & 1) swap(e[10], e[11]);
    int s = 24;
    for (int i = 0; i < 7; i++) {
      int a = (int)(3 * myrand());
      s -= a;
      c[i] = corner_val(corner_perm(c[i]), a);
    }
    c[7] = corner_val(corner_perm(c[7]), s % 3);
    s = 0;
    for (int i = 0; i < 11; i++) {
      int a = (int)(2 * myrand());
      e[i] = edge_ori_add(e[i], a);
      s ⊕= a;
    }
    e[11] ⊕= s;
  }
```

69. The canonicalization into 96 is the same, but it selects the lesser of the two to start with, and then canonicalizes into the destination one after the other.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void cubepos::canon_into96(cubepos &dst) const
  {
    cubepos cpi;
    invert_into(cpi);
    if (*this < cpi) {
      dst = *this;
    }
    else {
      dst = cpi;
    }
    canon_into48_aux(dst);
    cpi.canon_into48_aux(dst);
  }
```

70. When exploring the state space recursively, it is import for efficiency to prune the search as early as possible. One way to do this is to be efficient in the move sequences we explore. For instance, it is never advantageous to consider a sequence that has two consecutive turns of the same face, in the half turn metric, because such a position can always be obtained from a shorter sequence. Furthermore, consecutive rotations of opposite faces should always occur in the same order (we choose earlier numbered faces first).

```
⟨ Global utility declarations 30 ⟩ +≡
  const int CANONSEQSTATES = FACES + 1;
  const int CANONSEQSTART = 0;
```

71. We need a couple of small arrays to give us the next state and the bit mask of allowed moves.

```
⟨ Static data declarations of cubepos 12 ⟩ +≡
  static unsigned char canon_seq[CANONSEQSTATES][NMOVES];
  static int canon_seq_mask[CANONSEQSTATES];
  static int canon_seq_mask_ext[CANONSEQSTATES];
```

72. We instantiate these arrays.

```
⟨ Static data instantiations 13 ⟩ +≡
  unsigned char cubepos::canon_seq[CANONSEQSTATES][NMOVES];
  int cubepos::canon_seq_mask[CANONSEQSTATES];
  int cubepos::canon_seq_mask_ext[CANONSEQSTATES];
```

73. Initializing these arrays is pretty easy based on the rules we have outlined. In the halfturn metric, the state is just one plus the previous face that was twisted.

```
⟨ Initialization of cubepos 14 ⟩ +≡
  for (int s = 0; s < CANONSEQSTATES; s++) {
    int prevface = (s − 1) % FACES;
    canon_seq_mask[s] = (1 ≪ NMOVES) − 1;
    for (int mv = 0; mv < NMOVES; mv++) {
      int f = mv/TWISTS;
      int isplus = 0;
      if (s ≠ 0 ∧ (prevface ≡ f ∨ prevface ≡ f + 3))      /* illegal */
      {
        canon_seq[s][mv] = INVALID;
        canon_seq_mask[s] &= ∼(1 ≪ mv);
      }
      else {
        canon_seq[s][mv] = f + 1 + FACES * isplus;
      }
    }
    canon_seq_mask_ext[s] = canon_seq_mask[s];
  }
```

74. The utility routines to access these arrays.

```
⟨ Public method declarations of cubepos 9 ⟩ +≡
  static inline int next_cs(int cs, int mv)
  {
    return canon_seq[cs][mv];
  }
  static inline int cs_mask(int cs)
  {
    return canon_seq_mask[cs];
  }
  static inline int cs_mask_ext(int cs)
  {
    return canon_seq_mask_ext[cs];
  }
```

75. We finish with a number of generic utility routines for cube programming, such as error reporting, calculating the duration between two points, and random number generation.

```
⟨ Global utility declarations 30 ⟩ +≡
  void
  error (const char *s) ;
  inline double myrand( )
  {
    return drand48( );
  }
  inline int random_move( )
  {
    return (int)(NMOVES * myrand( ));
  }
  inline int random_move_ext( )
  {
    return (int)(NMOVES * myrand( ));
  }
  double walltime( );
  double duration( );
```

76. Implementing these methods is straightforward.

```
⟨ cubepos.cpp 11 ⟩ +≡
  void error (const char *s)
  {
    cerr ≪ s ≪ endl;
    if (*s ≡ '!') exit(10);
  }
  static double start;
  double walltime( )
  {
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + 0.000001 * tv.tv_usec;
  }
  double duration( )
  {
    double now = walltime( );
    double r = now − start;
    start = now;
    return r;
  }
```

77. Testing. This class is no good to us if it doesn't work, and what better way to verify it works than in isolation. This program tests cubepos, and also provides an example of how to use it.

```
⟨ cubepos_test.cpp 77 ⟩ ≡
#include <iostream>
#include <map>
#include "cubepos.h"
  void check(const cubepos &cp1, const cubepos &cp2, const char *msg)
  {
    if (cp1 ≡ cp2) return;
    for (int i = 0; i < 8; i++) cout ≪ "␣" ≪ (int)(cp1.c[i]) ≪ "␣" ≪ (int)(cp2.c[i]) ≪ endl;
    for (int i = 0; i < 12; i++) cout ≪ "␣" ≪ (int)(cp1.e[i]) ≪ "␣" ≪ (int)(cp2.e[i]) ≪ endl;
    cout ≪ endl ≪ msg ≪ endl;
    exit(10);
  }
```

See also sections 78, 79, 80, and 81.

78. We define a recursive routine that just fills an array with all positions seen from a depth-first search. This is a common search paradigm.

```
⟨ cubepos_test.cpp 77 ⟩ +≡
  void recur1(const cubepos &cp, int togo, int canonstate, vector⟨cubepos⟩ &a)
  {
    a.push_back(cp);
    if (togo−−) {
      cubepos cp2;
      int mask = cubepos::cs_mask(canonstate);
      for (int mv = 0; mv < NMOVES; mv++) {
        if ((mask ≫ mv) & 1) {
          cp2 = cp;
          cp2.move(mv);
          recur1(cp2, togo, cubepos::next_cs(canonstate, mv), a);
        }
      }
    }
  }
```

79. Some of our tests generate known results on the count of positions at each depth. These arrays hold the known results for comparison.

```
⟨ cubepos_test.cpp 77 ⟩ +≡
  unsigned int allpos[] = {1, 18, 243, 3240, 43239, 574908, 7618438, 100803036, 1332343288};
  unsigned int c48pos[] = {1, 2, 9, 75, 934, 12077, 159131, 2101575, 27762103, 366611212};
  unsigned int c96pos[] = {1, 2, 8, 48, 509, 6198, 80178, 1053077, 13890036, 183339529};
```

80. We need random move sequences sometimes.

```
⟨ cubepos_test.cpp 77 ⟩ +≡
  moveseq random_moveseq(int len)
  {
    moveseq r;
    for (int i = 0; i < len; i++) r.push_back(random_move());
    return r;
  }
```

81. The main routine. We simply execute the tests sequentially.

```
⟨ cubepos_test.cpp 77 ⟩ +≡
  const unsigned int MAXELEMENTS = 100000;
  map⟨cubepos, int⟩ world;
  vector⟨cubepos⟩ q;
  int main(int argc, char *argv[])
  {
    cubepos cp, cp2, cp3, cp4;
    ⟨ Basic tests 82 ⟩
    ⟨ Move tests 83 ⟩
    ⟨ Inversion tests 84 ⟩
    ⟨ Multiplication tests 85 ⟩
    ⟨ Move parsing tests 86 ⟩
    ⟨ Singmaster tests 87 ⟩
    ⟨ Symmetry tests 88 ⟩
    ⟨ Breadth-first search one. 89 ⟩
    ⟨ Breadth-first search two. 90 ⟩
    ⟨ Breadth-first search three. 91 ⟩
    ⟨ Depth-first search one. 92 ⟩
  }
```

82. Basic tests: size, set up random number generator, check identity cube. We also verify that the first sixteen remappings preserve the up/down faces.

```
⟨ Basic tests 82 ⟩ ≡
  if (sizeof(int) ≠ 4)
    error ("! this code assumes a 4-byte int throughout") ;
  if (sizeof(short) ≠ 2)
    error ("! this code assumes a two-byte short") ;
  if (sizeof(cubepos) ≠ 20)
    error ("! size of cubepos is not 20") ;
  if (lrand48() ≡ 0) srand48(getpid() + time(0));
  for (int i = 0; i < 8; i++)
    if (cp.c[i] ≠ identity_cube.c[i])
      error ("! bad initial cp") ;
  for (int i = 0; i < 12; i++)
    if (cp.e[i] ≠ identity_cube.e[i])
      error ("! bad initial cp") ;
  for (int i = 0; i < 16; i++)
    if (cubepos::face_map[i][0] % 3 ≠ 0)
      error ("! up down not preserved in first 16") ;
```

This code is used in section 81.

83. Testing the moves. We verify that every clockwise move has order 4. We also test that from the solved position, making a move with *movepc* undoes a move made with *move*. This tests the two move routines against each other.

```
⟨ Move tests 83 ⟩ ≡
  cout ≪ "Verifying␣f/b␣moves." ≪ endl;
  for (int i = 0; i < NMOVES; i++) {
    cp.move(i);
    cp.movepc(i);
    check(cp, identity_cube, "problem␣verifying␣fb␣of␣moves");
  }
  cout ≪ "Verifying␣forward␣move." ≪ endl;
  for (int i = 0; i < FACES; i++) {
    for (int j = 0; j < 4; j++) cp.move(i * TWISTS);
    check(cp, identity_cube, "problem␣verifying␣order␣of␣basic␣generators");
  }
  cout ≪ "Verifying␣bw␣moves." ≪ endl;
  for (int i = 0; i < FACES; i++) {
    for (int j = 0; j < 4; j++) cp.movepc(i * TWISTS);
    check(cp, identity_cube, "problem␣verifying␣order␣of␣basic␣generators␣2");
  }
```

This code is used in section 81.

84. We test that the basic inversion routine works: that we can invert a cube twice and end up with the original cube. We also test generating a random sequence of moves, inverting the sequence, and then verify that applying the original sequence to a cube is the same as applying the inverted sequence and then inverting the result.

```
⟨ Inversion tests 84 ⟩ ≡
  cout ≪ "Random␣cube␣inversion" ≪ endl;
  for (int i = 0; i < 100; i++) {
    cp.randomize( );
    cp.invert_into(cp2);
    cp2.invert_into(cp3);
    check(cp, cp3, "Inversion␣failed.");
  }
  cout ≪ "Move␣inversion" ≪ endl;
  for (int i = 0; i < 100; i++) {
    moveseq ms = random_moveseq(10);
    moveseq msi = cubepos::invert_sequence(ms);
    cp = identity_cube;
    cp2 = identity_cube;
    for (unsigned int k = 0; k < ms.size( ); k++) {
      cp.move(ms[k]);
      cp2.move(msi[k]);
    }
    cp.invert_into(cp3);
    check(cp2, cp3, "Invert␣move␣sequence␣failed");
  }
```

This code is used in section 81.

85. Next we test that the group multiplication operations work, by comparing the result of appended sequences against the result of direct multiplication of positions resulting from the original component sequences. We do this for both *move* and *movepc*.

```
⟨ Multiplication tests 85 ⟩ ≡
  cout ≪ "Multiplication" ≪ endl;
  for (int i = 0; i < 100; i++) {
    moveseq ms = random_moveseq(10), ms2 = random_moveseq(10);
    cp = identity_cube;
    cp2 = identity_cube;
    cp3 = identity_cube;
    for (unsigned int k = 0; k < ms.size(); k++) {
      cp.move(ms[k]);
      cp3.move(ms[k]);
    }
    for (unsigned int k = 0; k < ms2.size(); k++) {
      cp2.move(ms2[k]);
      cp3.move(ms2[k]);
    }
    cubepos::mul(cp, cp2, cp4);
    check(cp4, cp3, "Bad product");
    cp = identity_cube;
    cp2 = identity_cube;
    cp3 = identity_cube;
    for (unsigned int k = 0; k < ms.size(); k++) {
      cp.movepc(ms[k]);
      cp3.movepc(ms[k]);
    }
    for (unsigned int k = 0; k < ms2.size(); k++) {
      cp2.movepc(ms2[k]);
      cp3.movepc(ms2[k]);
    }
    cubepos::mulpc(cp, cp2, cp4);
    check(cp4, cp3, "Bad product");
  }
```

This code is used in section 81.

86. Next we check that we can serialize a move sequence and parse it back in and get the same result.

```
⟨ Move parsing tests 86 ⟩ ≡
  cout ≪ "Test parse move" ≪ endl;
  for (int i = 0; i < 100; i++) {
    moveseq ms = random_moveseq(10);
    char movebuf[1000];
    char *p = movebuf;
    for (unsigned int j = 0; j < ms.size(); j++) cubepos::append_move(p, ms[j]);
    const char *pp = movebuf;
    moveseq ms2 = cubepos::parse_moveseq(pp);
    if (ms ≠ ms2)
      error ("! bad parse") ;
  }
```

This code is used in section 81.

87. We next check that the Singmaster position parsing and printing routines are proper inverses of each other.

```
⟨ Singmaster tests 87 ⟩ ≡
  cout ≪ "Testing␣Singmaster" ≪ endl;
  for (int i = 0; i < 100; i++) {
    char singbuf[1000];
    cp.randomize();
    strcpy(singbuf, cp.Singmaster_string());
    const char *err = cp2.parse_Singmaster(singbuf);
    if (err)
      error (err) ;
    check(cp, cp2, "!␣mismatch␣between␣parse␣and␣gen");
  }
```

This code is used in section 81.

88. We next check out our remapping routines; we verify that a random move sequence applied, and then the position remapped, is the same as applying the remapped sequence.

```
⟨ Symmetry tests 88 ⟩ ≡
  cout ≪ "Testing␣remap" ≪ endl;
  for (int i = 0; i < 100; i++) {
    moveseq ms;
    int m = (int)(M * myrand());
    for (int j = 0; j < 1; j++) ms.push_back(random_move());
    cp = identity_cube;
    cp2 = identity_cube;
    for (unsigned int j = 0; j < ms.size(); j++) {
      cp.move(ms[j]);
      cp2.move(cubepos::move_map[m][ms[j]]);
    }
    cp.remap_into(m, cp3);
    check(cp2, cp3, "Move␣map␣issue");
  }
```

This code is used in section 81.

89. Our first breadth-first search inserts all positions.

```
⟨ Breadth-first search one. 89 ⟩ ≡
  world.clear( );
  q.clear( );
  q.push_back(identity_cube);
  world[identity_cube] = 0;
  unsigned int qg = 0;
  int prevd = −1;
  duration( );
  while (qg < q.size( )) {
    int d = world[q[qg]];
    if (d ≠ prevd) {
      cout ≪ "At lev " ≪ d ≪ " size " ≪ (q.size( ) − qg) ≪ endl;
#ifndef SLICE
      if (allpos[d] ≠ q.size( ) − qg)
        error ("! bad value") ;
#endif
      if (q.size( ) > MAXELEMENTS) break;
      prevd = d;
    }
    for (int i = 0; i < NMOVES; i++) {
      cp = q[qg];
      cp.move(i);
      if (world.find(cp) ≡ world.end( )) {
        world[cp] = d + 1;
        q.push_back(cp);
      }
    }
    qg++;
  }
  cout ≪ "Took " ≪ duration( ) ≪ endl;
```

This code is used in section 81.

90. Our second breadth-first search only inserts canonical positions.

```
⟨ Breadth-first search two. 90 ⟩ ≡
  world.clear( );
  q.clear( );
  q.push_back(identity_cube);
  world[identity_cube] = 0;
  qg = 0;
  prevd = −1;
  while (qg < q.size( )) {
    int d = world[q[qg]];
    if (d ≠ prevd) {
      cout ≪ "At␣lev␣" ≪ d ≪ "␣size␣" ≪ (q.size( ) − qg) ≪ endl;
#ifndef SLICE
      if (c48pos[d] ≠ q.size( ) − qg)
        error ("!␣bad␣value") ;
#endif
      if (q.size( ) > MAXELEMENTS) break;
      prevd = d;
    }
    for (int i = 0; i < NMOVES; i++) {
      cp = q[qg];
      cp.move(i);
      cp.canon_into48(cp2);
      if (world.find(cp2) ≡ world.end( )) {
        world[cp2] = d + 1;
        q.push_back(cp2);
      }
    }
    qg++;
  }
```

This code is used in section 81.

91. Our third breadth-first search only inserts canonical (mod M and inverse) positions. Note that we need to do both moves and premoves in this case.

```
⟨ Breadth-first search three. 91 ⟩ ≡
  cout ≪ "Took␣" ≪ duration() ≪ endl;
  world.clear();
  q.clear();
  q.push_back(identity_cube);
  world[identity_cube] = 0;
  qg = 0;
  prevd = −1;
  while (qg < q.size()) {
    int d = world[q[qg]];
    if (d ≠ prevd) {
      cout ≪ "At␣lev␣" ≪ d ≪ "␣size␣" ≪ (q.size() − qg) ≪ endl;
#ifndef SLICE
      if (c96pos[d] ≠ q.size() − qg)
        error ("!␣bad␣value") ;
#endif
      if (q.size() > MAXELEMENTS) break;
      prevd = d;
    }
    for (int i = 0; i < NMOVES; i++) {
      cp = q[qg];
      cp.move(i);
      cp.canon_into96(cp2);
      if (world.find(cp2) ≡ world.end()) {
        world[cp2] = d + 1;
        q.push_back(cp2);
      }
      cp = q[qg];
      cp.movepc(i);
      cp.canon_into96(cp2);
      if (world.find(cp2) ≡ world.end()) {
        world[cp2] = d + 1;
        q.push_back(cp2);
      }
    }
    qg++;
  }
  cout ≪ "Took␣" ≪ duration() ≪ endl;
```

This code is used in section 81.

92. Our depth-first search uses the earlier recursive routine. We print out the durations of the phases; on my computer, the sort phase dominates.

```
⟨ Depth-first search one. 92 ⟩ ≡
  world.clear( );
  unsigned int prevcount = 0;
  for (int d = 0; ; d++) {
    q.clear( );
    double t1 = walltime( );
    recur1(identity_cube, d, CANONSEQSTART, q);
    double t2 = walltime( );
    sort(q.begin( ), q.end( ));
    double t3 = walltime( );
    vector⟨cubepos⟩::iterator nend = unique(q.begin( ), q.end( ));
    double t4 = walltime( );
    unsigned int sz = nend − q.begin( );
    cout ≪ "Sequences␣" ≪ q.size( ) ≪ "␣positions␣" ≪ sz ≪ endl;
    cout ≪ "At␣lev␣" ≪ d ≪ "␣size␣" ≪ (sz − prevcount) ≪ endl;
    cout ≪ "Search␣" ≪ (t2 − t1) ≪ "␣sort␣" ≪ (t3 − t2) ≪ "␣uniq␣" ≪ (t4 − t3) ≪ endl;
#ifndef SLICE
    if (allpos[d] ≠ sz − prevcount)
      error ("!␣bad␣value") ;
#endif
    prevcount = sz;
    if (sz > 3000000) break;
  }
  cout ≪ "Took␣" ≪ duration( ) ≪ endl;
```

This code is used in section 81.

⟨ Basic tests 82 ⟩ Used in section 81.
⟨ Breadth-first search one. 89 ⟩ Used in section 81.
⟨ Breadth-first search three. 91 ⟩ Used in section 81.
⟨ Breadth-first search two. 90 ⟩ Used in section 81.
⟨ Data representation of cubepos 7, 8 ⟩ Used in section 5.
⟨ Depth-first search one. 92 ⟩ Used in section 81.
⟨ Global utility declarations 30, 64, 70, 75 ⟩ Used in section 5.
⟨ Initialization of cubepos 14, 28, 29, 34, 51, 60, 61, 62, 73 ⟩ Used in section 11.
⟨ Inversion tests 84 ⟩ Used in section 81.
⟨ Local routines for cubepos 36, 50, 53, 58, 59 ⟩ Used in section 11.
⟨ Move parsing tests 86 ⟩ Used in section 81.
⟨ Move tests 83 ⟩ Used in section 81.
⟨ Multiplication tests 85 ⟩ Used in section 81.
⟨ Public method declarations of cubepos 9, 10, 15, 20, 31, 37, 41, 43, 52, 63, 67, 74 ⟩ Used in section 5.
⟨ Singmaster tests 87 ⟩ Used in section 81.
⟨ Static data declarations of cubepos 12, 18, 21, 32, 55, 71 ⟩ Used in section 5.
⟨ Static data instantiations 13, 19, 22, 24, 25, 26, 27, 33, 44, 47, 48, 49, 56, 57, 72 ⟩ Used in section 11.
⟨ Static initialization hack 16 ⟩ Used in section 5.
⟨ Symmetry tests 88 ⟩ Used in section 81.
⟨ `cubepos.cpp` 11, 17, 23, 35, 38, 39, 40, 42, 45, 46, 54, 65, 66, 68, 69, 76 ⟩
⟨ `cubepos.h` 2, 4, 5 ⟩
⟨ `cubepos_test.cpp` 77, 78, 79, 80, 81 ⟩

CUBEPOS