1. Introduction. The **cubepos** package provides a rich and fast representation of the Rubik's cube, with a full set of operations including moves, multiplication, and inversion. (Please read that document if you haven't already, because **kocsymm** builds on that.) While **cubepos** is rich and efficient, sometimes it is not exactly what you need. For instance, if you wanted to use the **cubepos** structure to create an index into an array based on the 12! possible permutations of the edges, you would have to do a fair amount of work. Other representations of the cube provide faster indexing operations. The two classes defined here, **kocsymm** and **permcube**, provide an alternative representation of the cube that is particularly suitable for implementations of Herbert Kociemba's two-phase algorithm.

⟨ `kocsymm.h` 1 ⟩ ≡

```
#ifndef KOCSYMM_H
#define KOCSYMM_H
#include "cubepos.h"
```

See also sections 2, 4, 29, 30, and 55.

2. The two-phase algorithm is based on the subgroup generated by $\{U, F2, R2, D, B2, L2\}$. We refer to this group as H. The idea behind the two-phase algorithm is to find a way to take an arbitrary cube position and find a sequence to bring it into the subgroup, and then solve it within the subgroup using moves within the subgroup. The first phase is just finding a path within the Schreier coset graph of the group H to the trivial coset; the second part is solving within that coset.

The orientation conventions in **cubepos** were carefully chosen so that none of the moves that generate H change the orientation of any of the cubies in the solved position; thus, all positions in H, when represented by **cubepos**, have the same orientation (both edge and corners) as the solved position. Furthermore, none of the moves that generate H move any of the cubies in the middle slice out of the middle slice; thus, this is also preserved for all positions in H.

It turns out (but we will not prove it here) that every position that meets these conditions is in the subgroup H. Each of the 8! permutations of the corners is reachable, in combination with each of the 8! permutations of the top and bottom edges, and also in combination with all 4! permutations of the middle edges subject that the overall parity of the corners and edges match. Thus, the size of H is $8! \cdot 8! \cdot 4!/2$ or 19,508,428,800.

Given a representative position p, the right coset of H corresponding to p is just Hp (where the multiplication operation of the group is extended to sets in the usual way). Since all the positions in H have the same, solved, orientation, this means all the positions in any particular coset of H (identified by p) share the same orientation. Similarly, the set of cubie slots that contain middle edge cubies is also the same for every element of a particular coset of H. Indeed, these three things, the edge orientation, the corner orientation, and the cubie slots containing the middle edge cubies, fully define a coset of H.

The **kocsymm** class contains three integer coordinates that each represent one of these characteristics, and thus, an entire coset of H. The *move* operation on a **kocsymm** instance is just an edge on the coset graph of H induced by the set of move generators we have chosen. By using simple integer coordinates, we allow for very fast operations of *move* and indexing. The *csymm* coordinate represents the corner orientation and has 3^7 possible values; the *eosymm* coordinate represents the edge orientation and has 2^{11} possible values; the *epsymm* coordinate has $\binom{12}{4}$ possible values and represents the slots that contain the middle four slice cubies. Where order matters, we will always give the coordinates in this order. (They are ordered from largest range to smallest range.) We define a type that is large enough for these ranges (and also 8!, incidentally) yet still storage efficient, and use this type for most of our tables. For the trivial coset, the value of all three coordinates is chosen to be zero.

```
⟨ kocsymm.h 1 ⟩ +≡
  const int CORNERSYMM = 2187;
  const int EDGEOSYMM = 2048;
  const int EDGEPERM = 495;
  ⟨ Constants for kocsymm and permcube 18 ⟩
  typedef unsigned short lookup_type;
  class kocsymm {
  public:
    kocsymm( )
    : csymm(0), eosymm(0), epsymm(0) { }
    kocsymm(int c, int eo, int ep)
    : csymm(c), eosymm(eo), epsymm(ep) { }
    ⟨ Methods for kocsymm 3 ⟩
    ⟨ Static data declarations for kocsymm 6 ⟩
    lookup_type csymm, eosymm, epsymm;
  };
```

3. We have the same static initialization issue with **kocsymm** that we did with **cubepos**, so we declare a special initializer that forces an initialization routine to be called, as well as that initialization routine itself.

```
⟨ Methods for kocsymm 3 ⟩ ≡
  kocsymm(int)
  : csymm(0), eosymm(0), epsymm(0) {
    init( );
  }
  static void init( );
```

See also sections 5, 9, 10, 24, 26, and 28.

This code is used in section 2.

4. To force initialization in the proper order for all users of this include file, we declare a file-static instance here. Since we need an identity anyway, we go ahead and make this the identity object for this class (although there will be multiple instances, they will all have the same value).

```
⟨ kocsymm.h 1 ⟩ +≡
  static kocsymm identity_kc(1);
```

5. We use simple ordering, equality, and inequality methods for this simple value class.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  inline bool operator < (const kocsymm &kc) const
  {
    if (csymm ≠ kc.csymm) return csymm < kc.csymm;
    if (eosymm ≠ kc.eosymm) return eosymm < kc.eosymm;
    return epsymm < kc.epsymm;
  }
  inline bool operator≡(const kocsymm &kc) const
  {
    return kc.csymm ≡ csymm ∧ kc.eosymm ≡ eosymm ∧ kc.epsymm ≡ epsymm;
  }
  inline bool operator≠(const kocsymm &kc) const
  {
    return kc.csymm ≠ csymm ∨ kc.eosymm ≠ eosymm ∨ kc.epsymm ≠ epsymm;
  }
```

6. We want to implement a fast move operation, and the range of each of the coordinates is fairly small, so we use static tables to implement *move*. We choose to use the coordinate value as the first index, as many of our searches will be depth-first search, and this will give us some cache locality we would otherwise lose out on.

```
⟨ Static data declarations for kocsymm 6 ⟩ ≡
  static lookup_type cornermove[CORNERSYMM][NMOVES];
  static lookup_type edgeomove[EDGEOSYMM][NMOVES];
  static lookup_type edgepmove[EDGEPERM][NMOVES];
```

See also sections 11 and 20.

This code is used in section 2.

7. These arrays need to be allocated, so it is time to introduce our *cpp* source.

```
⟨ kocsymm.cpp 7 ⟩ ≡
#include "kocsymm.h"
#include <iostream>
  using namespace std;
  ⟨ Static data instantiations 8 ⟩
  ⟨ Utility methods 13 ⟩
  ⟨ Method bodies 15 ⟩
  void kocsymm::init( )
  {
    static int initialized = 0;
    if (initialized) return;
    initialized = 1;
    ⟨ Initialize kocsymm 14 ⟩
    permcube::init( );
  }
```

8. We need to instantiate these arrays.

```
⟨ Static data instantiations 8 ⟩ ≡
  lookup_type kocsymm::cornermove[CORNERSYMM][NMOVES];
  lookup_type kocsymm::edgeomove[EDGEOSYMM][NMOVES];
  lookup_type kocsymm::edgepmove[EDGEPERM][NMOVES];
```

See also sections 12, 21, 32, 37, 40, and 44.

This code is used in section 7.

9. With these arrays, the *move* operation is very simple.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  void move(int mv)
  {
    csymm = cornermove[csymm][mv];
    eosymm = edgeomove[eosymm][mv];
    epsymm = edgepmove[epsymm][mv];
  }
```

10. The easiest way to initialize these arrays are to introduce conversion routines that allow us to extract the coordinates from a **cubepos**, and allow us to set up a **cubepos** with those characteristics. We can use a constructor to go from **cubepos** to **kocsymm**, but we use a *set_coset* to modify an existing **cubepos** so it is in the coset represented by the current **kocsymm**.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  kocsymm(const cubepos &cp);
  void set_coset(cubepos &cp);
```

11. Numbering the coordinates. For the corner symmetries, the easiest numbering representation is just as base-3 number, where the least significant digit comes from corner 0, and so on, and with the value from corner 7 ignored (since it must the the negative sum of the other corners). Similarly, the edge symmetries are most easily handled as a base-2 number from the first 11 edges.

The slots holding middle edge cubies is just a bit more complicated. We insist that the zero value be the solved position. First we build a bitmask that always has four bits set; the least significant four bits represent edge slots 4 to 7 (the middle slots), the next four bits represent edge slots 8 to 11, and the final four bits represent edge slots 0 to 3. We sort all possible 12-bit values in increasing numerical value, and use the index into this array to determine the value for *epsymm*.

To support this, we need two arrays, one to compress the bits from 12 bits down to an *epsymm* value, and one to expand the *epsymm* back into a bitmask. The rotations are done in the arrays, so the values you will obtain from the array and/or pass into the array all have the bits in normal, 0 though 12, order.

```
⟨ Static data declarations for kocsymm 6 ⟩ +≡
  static lookup_type epsymm_compress[1 ≪ 12];
  static lookup_type epsymm_expand[EDGEOSYMM];
```

12. The usual instantiation.

```
⟨ Static data instantiations 8 ⟩ +≡
  lookup_type kocsymm::epsymm_compress[1 ≪ 12];
  lookup_type kocsymm::epsymm_expand[EDGEOSYMM];
```

13. To help us fill these arrays, we need a generic bit counting function. This is not used in any performance-critical code, so we can be a bit slow.

```
⟨ Utility methods 13 ⟩ ≡
  static int bc(int v)
  {
    int r = 0;
    while (v) {
      v &= v − 1;
      r++;
    }
    return r;
  }
```

See also section 35.

This code is used in section 7.

14. Filling these two arrays is straightforward. We also fill the entry without the high bit set, just in case we decide to only look at 11 cubies rather than 12.

```
⟨ Initialize kocsymm 14 ⟩ ≡
  int c = 0;
  for (int i = 0; i < 1 ≪ 12; i++)
    if (bc(i) ≡ 4) {
      int rotval = ((i ≪ 4) + (i ≫ 8)) & #fff;
      epsymm_compress[rotval] = c;
      epsymm_compress[rotval & #7ff] = c;
      epsymm_expand[c] = rotval;
      c++;
    }
```

See also sections 17, 22, and 23.

This code is used in section 7.

15. With that done, we are now ready to obtain a **kocsymm** object from a **cubepos**. This routine does not have to be dramatically fast. We use a little trick; of the edge indices 0 through 11, only those in the middle edge, with values 4 though 7, have the bit with value 4 set. Since the cubie numbering for edges has the orientation in the low bit, this means we actually need to use the bit with value 8.

⟨ Method bodies 15 ⟩ ≡

```
kocsymm::kocsymm(const cubepos &cp)
{
  int c = 0, eo = 0, ep = 0;
  for (int i = 6; i ≥ 0; i--) c = 3 * c + cubepos::corner_ori(cp.c[i]);
  for (int i = 10; i ≥ 0; i--) {
    eo = 2 * eo + cubepos::edge_ori(cp.e[i]);
    ep = 2 * ep + (cp.e[i] & 8);
  }
  csymm = c;
  eosymm = eo;
  epsymm = epsymm_compress[ep ≫ 3];
}
```

See also sections 16, 25, 27, 33, 46, 48, 49, 50, 51, and 52.

This code is used in section 7.

16. Setting a cubepos to be in the coset is also straightforward. We completely destroy the pre-existing permutation in the **cubepos** as we do this. This routine is not particularly fast. The only complexity in this routine is recovering the orientation of the last corner and edge.

```
⟨ Method bodies 15 ⟩ +≡
  void kocsymm::set_coset(cubepos &cp)
  {
    int c = csymm, eo = eosymm, ep = epsymm_expand[epsymm];
    int s = 0;
    for (int i = 0; i < 7; i++) {
      int ori = c % 3;
      cp.c[i] = cubepos::corner_val(i, ori);
      s += ori;
      c = c/3;
    }
    cp.c[7] = cubepos::corner_val(7, (8 * 3 - s) % 3);
    s = 0;
    int nextmid = 4;
    int nextud = 0;
    for (int i = 0; i < 12; i++) {
      if (i ≡ 11) eo = s;
      int ori = eo & 1;
      if (ep & 1) cp.e[i] = cubepos::edge_val(nextmid++, ori);
      else {
        cp.e[i] = cubepos::edge_val(nextud++, ori);
        if (nextud ≡ 4) nextud = 8;
      }
      s ⊕= ori;
      eo ≫= 1;
      ep ≫= 1;
    }
  }
```

17. With these two routines in place, we can fill out our move arrays. Note that we have to use *movepc*, since the coset space (which is not a group) doesn't know where the cubies are, only what the orientations are in specific slots (and a bit more information about the middle cubies).

```
⟨ Initialize kocsymm 14 ⟩ +≡
  cubepos cp, cp2;
  for (int i = 0; i < CORNERSYMM; i++) {
    kocsymm kc(i, i % EDGEOSYMM, i % EDGEPERM);
    kc.set_coset(cp);
    for (int mv = 0; mv < NMOVES; mv++) {
      cp2 = cp;
      cp2.movepc(mv);
      kocsymm kc2(cp2);
      cornermove[i][mv] = kc2.csymm;
      if (i < EDGEOSYMM) edgeomove[i][mv] = kc2.eosymm;
      if (i < EDGEPERM) edgepmove[i][mv] = kc2.epsymm;
    }
  }
```

18. Symmetry. Just like **cubepos**, **kocsymm** (and later, **permcube**) have symmetry. Since these treat the middle layer cubies differently than the others, some symmetry is broken; we only have 16-way rather than 48-way symmmetry. The symmetry of **kocsymm** and **permcube** are specifically the first 16 symmetries of **cubepos**, which was carefully constructed so the middle cubies remain middle cubies.

Calculating corner orientation remapping and middle edge slot remapping is straightforward, but edge orientation remapping is not so simple. Our edge orientation convention as defined by **cubepos** treats the front and back faces differently from the left and right faces, so we cannot simply shuffle the bits around. However, the first 8 symmetries of **cubepos** preserve all three axes, so we can just shuffle bits for the first eight symmetries. For the other eight symmetries, we can use the *epsymm* information to determine which cubies are out of the middle slice both before and after rotation, and exclusive-or that information to correct for this rotation.

In reality, the only thing we use the rotation for is to canonicalize a **kocsymm**, so we do not store full remapping information for the corner symmetry, only enough information for canonicalization. As part of this canonicalization we also squeeze and unsqueeze the corner coordinate (to eliminate gaps created by the canonicalization). We define `CORNERRSYMM` to represent the count of canonical corner permutations, which we precompute and put here, and then check later in the initialization routine.

```
⟨ Constants for kocsymm and permcube 18 ⟩ ≡
  const int KOCSYMM = 16;
  const int CORNERRSYMM = 168;
```

See also section 19.

This code is used in section 2.

19. We need a set of arrays to manage the canonicalization. We need remapping arrays for the edge orientation and permutation. We need an array for the edge permutation that says what bits to flip (but we only need one entry, used only if we are remapping to 8 through 15). For corner remapping, we have two cases. The most common case is there is a unique remapping that minimizes the corner coordinate, in which case canonicalization is quick and easy. The other case is when there are multiple distinct remappings that all generate the same minimal corner coordinate. In this case, we store a bitmask indicating which remappings to consider, and we must iterate through them all.

From the corner coordinate, we compute three data items: *minbits*, a set of 16 bits, one per symmetry, that generates the minimum corner coordinate value; *csymm*, the corner symm we get as a result (after compaction), and *mimap*, the minimum mapping that generates that value.

```
⟨ Constants for kocsymm and permcube 18 ⟩ +≡
  struct corner_mapinfo {
    unsigned short minbits;
    unsigned char csymm, minmap;
  };
```

20. We need the following arrays to support canonicalization.

```
⟨ Static data declarations for kocsymm 6 ⟩ +≡
  static lookup_type cornersymm_expand[CORNERRSYMM];
  static corner_mapinfo cornersymm[CORNERSYMM];
  static lookup_type edgeomap[EDGEOSYMM][KOCSYMM];
  static lookup_type edgepmap[EDGEPERM][KOCSYMM];
  static lookup_type edgepxor[EDGEPERM][2];
```

21. We need to instantiate those arrays.

```
⟨ Static data instantiations 8 ⟩ +≡
  lookup_type kocsymm :: cornersymm_expand[CORNERRSYMM];
  corner_mapinfo kocsymm :: cornersymm[CORNERSYMM];
  lookup_type kocsymm :: edgeomap[EDGEOSYMM][KOCSYMM];
  lookup_type kocsymm :: edgepmap[EDGEPERM][KOCSYMM];
  lookup_type kocsymm :: edgepxor[EDGEPERM][2];
```

22. Our strategy for initializing these is very similar to what we did for moves: use the **cubepos** class and the two conversion routines to do the heavy lifting. We start by figuring out the corner compaction values.

```
⟨ Initialize kocsymm 14 ⟩ +≡
  c = 0;
  for (int cs = 0; cs < CORNERSYMM; cs++) {
    int minval = cs;
    int lowm = 0;
    int lowbits = 1;
    kocsymm kc(cs, 0, 0);
    for (int m = 1; m < KOCSYMM; m++) {
      kc.set_coset(cp);
      cp.remap_into(m, cp2);
      kocsymm kc2(cp2);
      if (kc2.csymm < minval) {
        minval = kc2.csymm;
        lowbits = 1 ≪ m;
        lowm = m;
      }
      else if (kc2.csymm ≡ minval) {
        lowbits |= 1 ≪ m;
      }
    }
    if (minval ≡ cs) {
      cornersymm_expand[c] = minval;
      cornersymm[cs].csymm = c++;
    }
    cornersymm[cs].minbits = lowbits;
    cornersymm[cs].minmap = lowm;
    cornersymm[cs].csymm = cornersymm[minval].csymm;
  }
  if (c ≠ CORNERRSYMM)
    error ("!␣bad␣cornersym␣result") ;
```

23. Now we compute the edge permutation remapping, the xor values for the edge permutation, and the edge orientation remapping. Note that mapping 8 is self-inverse, so we reverse the result and input so we can apply the correction to *eosymm* before the array index.

```
⟨ Initialize kocsymm 14 ⟩ +≡
  for (int ep = 0; ep < EDGEPERM; ep++) {
    kocsymm kc(0, 0, ep);
    for (int m = 0; m < KOCSYMM; m++) {
      kc.set_coset(cp);
      cp.remap_into(m, cp2);
      kocsymm kc2(cp2);
      edgepmap[ep][m] = kc2.epsymm;
      if (m ≡ 8) {
        edgepxor[kc2.epsymm][0] = 0;
        edgepxor[kc2.epsymm][1] = kc2.eosymm;
      }
    }
  }
  for (int eo = 0; eo < EDGEOSYMM; eo++) {
    kocsymm kc(0, eo, 0);
    for (int m = 0; m < KOCSYMM; m++) {
      kc.set_coset(cp);
      cp.remap_into(m, cp2);
      kocsymm kc2(cp2);
      edgeomap[eo][m] = kc2.eosymm;
    }
  }
```

24. With these arrays, we are ready to canonicalize.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  void canon_into(kocsymm &kc) const;
```

25. The implementation first checks if we can do it quickly, and if not, iterates.

```
⟨ Method bodies 15 ⟩ +≡
  void kocsymm::canon_into(kocsymm &kc) const
  {
    corner_mapinfo &cm = cornersymm[csymm];
    kc.csymm = cornersymm_expand[cm.csymm];
    kc.eosymm = edgeomap[edgepxor[epsymm][cm.minmap ≫ 3] ⊕ eosymm][cm.minmap];
    kc.epsymm = edgepmap[epsymm][cm.minmap];
    for (int m = cm.minmap + 1; cm.minbits ≫ m; m++)
      if ((cm.minbits ≫ m) & 1) {
        int neo = edgeomap[edgepxor[epsymm][m ≫ 3] ⊕ eosymm][m];
        if (neo > kc.eosymm) continue;
        int nep = edgepmap[epsymm][m];
        if (neo < kc.eosymm ∨ nep < kc.epsymm) {
          kc.eosymm = neo;
          kc.epsymm = nep;
        }
      }
  }
```

26. We need a method that returns how much symmetry this **kocsymm** has.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  int calc_symm( ) const;
```

27. The implementation is just a slight rewriting of *canon_into*.

```
⟨ Method bodies 15 ⟩ +≡
  int kocsymm::calc_symm( ) const
  {
    int r = 1;
    corner_mapinfo &cm = cornersymm[csymm];
    int teosymm = edgeomap[edgepxor[epsymm][cm.minmap ≫ 3] ⊕ eosymm][cm.minmap];
    int tepsymm = edgepmap[epsymm][cm.minmap];
    for (int m = cm.minmap + 1; cm.minbits ≫ m; m++)
      if (((cm.minbits ≫ m) & 1) ∧ edgeomap[edgepxor[epsymm][m ≫ 3] ⊕ eosymm][m] ≡
          teosymm ∧ edgepmap[epsymm][m] ≡ tepsymm) r++;
    return r;
  }
```

28. We need a method that tells us if a move is in the Kociemba group or not. We can just determine if a transition from the default state of *epsymm* is zero or not.

```
⟨ Methods for kocsymm 3 ⟩ +≡
  static inline int in_Kociemba_group(int mv)
  {
    return edgepmove[0][mv] ≡ 0;
  }
```

29. Storing permutations with permcube. With **kocsymm** working, we can turn our attention to storing those bits of the state that are not stored in it—the permutation information. While **kocsymm** does store a limited amount of permutation information (what slots the middle four cubies are in), **permcube** stores all of the permutation information. We design **permcube** to enable fast moves and indexing of the resulting state, with the tradeoff that it is not as rich as **cubepos**; for instance, we do not define inversion.

We store edge permutation information and corner permutation information separately. The **kocsymm** class already defines the ability to maintain the position of four cubies at a time (as a group); we exploit that to maintain the slots for the upper edges and the lower edges as well. We store this information in the three fields *et*, *em*, and *eb* (edge top, edge middle, and edge bottom). For all three groups of four cubies, we store in addition the order that the cubies occur within that group of four; we store this in the fields *etp*, *emp*, and *ebp*. The information in *et*, *em*, and *eb* is redundant; if we know the slots holding either two sets, we also know the sets holding the other. Nonetheless, dividing the 12! or 479,001,600 possible states into six smaller chunks, three of 495 values and 3 of 24 values, makes our transition tables much smaller, and we share the same transition tables for the top, middle, and edge.

For the corners, we use a similar approach: we store which four of the eight slots contain top corner cubies in *c8_4*, and separately, we store the order of the top cubies in *ctp*, and the order of the bottom cubies in *cbp*.

```
⟨ kocsymm.h 1 ⟩ +≡
  const int FACT4 = 24;
  const int C8_4 = 70;
  class permcube {
  public:
    permcube( );
    ⟨ Methods for permcube 42 ⟩
    static void init( );
    ⟨ Static data declarations for permcube 31 ⟩
    unsigned short et, em, eb;
    unsigned char etp, emp, ebp;
    unsigned char c8_4, ctp, cbp;
  };
```

30. We allocate a file-scope identity instance statically. We don't actually need this one to work around the static initialization fiasco, but it's always good to have a cheap identity object.

```
⟨ kocsymm.h 1 ⟩ +≡
  static permcube identity_pc;
```

31. To manage all the permutations of four elements, we need to build the multiplication and inversion table for this group, called S_4. We also declare two arrays, one which takes an eight-byte value, two bits per element, that gives the permutation (the identity element would be $0b11100100$ or #e4; the least significant bits represent the first element) and gives the corresponding index for that permutation, and one that does the inverse of that.

```
⟨ Static data declarations for permcube 31 ⟩ ≡
  static unsigned char s4inv[FACT4];
  static unsigned char s4mul[FACT4][FACT4];
  static unsigned char s4compress[256];
  static unsigned char s4expand[FACT4];
```

See also sections 36, 39, and 43.

This code is used in section 29.

32. Next, we declare these.

⟨ Static data instantiations 8 ⟩ +≡
```
unsigned char permcube::s4inv[FACT4];
unsigned char permcube::s4mul[FACT4][FACT4];
unsigned char permcube::s4compress[256];
unsigned char permcube::s4expand[FACT4];
```

33. We need an initialization routine for **permcube**. This is called automatically by **kocsymm**::*init*( ) so we don't need a static initialization hack.

⟨ Method bodies 15 ⟩ +≡
```
void permcube::init( )
{
  ⟨ Initialize permcube 34 ⟩;
}
```

34. Permutation numbering. Normally we would number S_4 is lexicographical order. But for various reasons we need to compute the parity of the permutation quickly, so we use bit 0 of the indexing for that purpose; this avoids a table lookup.

We have another requirement, however, introduced by *hcoset*. Let $i(p)$ be the integer index assigned to permutation p, $p(i)$ to be the permutation associated with integer index i, and $a \cdot b$ to be the multiplication of the permutation a by the permutation b, and $j \oplus k$ to be the bit-wise exclusive-or of j and k. We want $p(i(a) \oplus 1) \cdot b = p(i(a \cdot b) \oplus 1)$. Essentially, we want to group our permutations into pairs, the first even and the second odd, such that right multiplication preserves the pairs. We need this so we can collect certain pairs of permutations into a 24-bit word, perform an operation on them, and be assured that the result will still fall into a single 24-bit word, rather than different halves of two different 24-bit words.

It turns out both of these are easy to arrange. We generate the permutations in lexicographical order, but use the inverse permutation rather than the forward permutation, and store the parity. For the c loop below, there are only two values left for c and d, so the two permutations generated in sequence will have these values swapped, which is precisely what we need. The parity is just the exclusive or of the least significant two bits of the lexicographical order index.

```
⟨ Initialize permcube 34 ⟩ ≡
  int cc = 0;
  for (int a = 0; a < 4; a++)
    for (int b = 0; b < 4; b++)
      if (a ≠ b)
        for (int c = 0; c < 4; c++)
          if (a ≠ c ∧ b ≠ c) {
            int d = 0 + 1 + 2 + 3 − a − b − c;
            int coor = cc ⊕ ((cc ≫ 1) & 1);
            int expanded = (1 ≪ (2 ∗ b)) + (2 ≪ (2 ∗ c)) + (3 ≪ (2 ∗ d));
            s4compress[expanded] = coor;
            s4expand[coor] = expanded;
            cc++;
          }
  for (int i = 0; i < FACT4; i++)
    for (int j = 0; j < FACT4; j++) {
      int k = s4compress[muls4(s4expand[i], s4expand[j])];
      s4mul[j][i] = k;
      if (k ≡ 0) s4inv[i] = j;
    }
```

See also sections 38, 41, 53, and 54.

This code is used in section 33.

35. We still need to write the *muls4* utility routine. This is simple enough that we simply extract the relevant bits inline.

```
⟨ Utility methods 13 ⟩ +≡
  int muls4(int a, int b)
  {
    int r = 3 & (b ≫ (2 ∗ (a & 3)));
    r += (3 & (b ≫ (2 ∗ ((a ≫ 2) & 3)))) ≪ 2;
    r += (3 & (b ≫ (2 ∗ ((a ≫ 4) & 3)))) ≪ 4;
    r += (3 & (b ≫ (2 ∗ ((a ≫ 6) & 3)))) ≪ 6;
    return r;
  }
```

36. For the edge groups of four, we use the same arrays as **kocsymm**; these have already been defined and initialized. For the corner groups, we need to write compaction and move arrays.

```
⟨ Static data declarations for permcube 31 ⟩ +≡
  static unsigned char c8_4_compact[256];
  static unsigned char c8_4_expand[C8_4];
  static unsigned char c8_4_parity[C8_4];
```

37. Next, we declare these.

```
⟨ Static data instantiations 8 ⟩ +≡
  unsigned char permcube::c8_4_compact[256];
  unsigned char permcube::c8_4_expand[C8_4];
  unsigned char permcube::c8_4_parity[C8_4];
```

38. To initialize these arrays, we again need to track the parity. The pattern is more complex for the eight-bit words that have four bits set, so we simply count inversions.

```
⟨ Initialize permcube 34 ⟩ +≡
  int c = 0;
  for (int i = 0; i < 256; i++)
    if (bc(i) ≡ 4) {
      int parity = 0;
      for (int j = 0; j < 8; j++)
        if (1 & (i ≫ j))
          for (int k = 0; k < j; k++)
            if (0 ≡ (1 & (i ≫ k))) parity++;
      c8_4_parity[c] = parity & 1;
      c8_4_compact[i] = c;
      c8_4_expand[c] = i;
      c++;
    }
```

39. The usual use for **permcube** is to handle operations within the Kociemba group H, where the middle edge positions are always in the middle edge. Thus, the group information for the top edges is just $\binom{8}{4}$ rather than $\binom{12}{4}$, so we need an array to compress the $\binom{12}{4}$ index (which ranges from 0 to 494) to a $\binom{8}{4}$ index.

```
⟨ Static data declarations for permcube 31 ⟩ +≡
  static unsigned char c12_8[EDGEPERM];
  static lookup_type c8_12[C8_4];
```

40. Next, we declare these.

```
⟨ Static data instantiations 8 ⟩ +≡
  unsigned char permcube::c12_8[EDGEPERM];
  lookup_type permcube::c8_12[C8_4];
```

41. Initializing this is straightforward; we expand the bits, remove the middle four, and compress them again.

```
⟨ Initialize permcube 34 ⟩ +≡
  for (int i = 0; i < EDGEPERM; i++) {
    int expbits = kocsymm::epsymm_expand[i];
    if (expbits & #0f0) c12_8[i] = 255;
    else {
      int ii = c8_4_compact[(expbits ≫ 4) + (expbits & 15)];
      c12_8[i] = ii;
      c8_12[ii] = i;
    }
  }
```

42. We need equality and ordering routines. These are a bit long because of the count of fields. Note that we cannot use *memcmp* reliably because there might be indeterminate padding.

```
⟨ Methods for permcube 42 ⟩ ≡
  inline bool operator < (const permcube &pc) const
  {
    if (et ≠ pc.et) return et < pc.et;
    if (em ≠ pc.em) return em < pc.em;
    if (eb ≠ pc.eb) return eb < pc.eb;
    if (etp ≠ pc.etp) return etp < pc.etp;
    if (emp ≠ pc.emp) return emp < pc.emp;
    if (ebp ≠ pc.ebp) return ebp < pc.ebp;
    if (c8_4 ≠ pc.c8_4) return c8_4 < pc.c8_4;
    if (ctp ≠ pc.ctp) return ctp < pc.ctp;
    return cbp < pc.cbp;
  }
  inline bool operator≡(const permcube &pc) const
  {
    return et ≡ pc.et ∧ em ≡ pc.em ∧ eb ≡ pc.eb ∧ etp ≡ pc.etp ∧ emp ≡ pc.emp ∧ ebp ≡ pc.ebp ∧ c8_4 ≡
        pc.c8_4 ∧ ctp ≡ pc.ctp ∧ cbp ≡ pc.cbp;
  }
  inline bool operator≠(const permcube &pc) const
  {
    return et ≠ pc.et ∨ em ≠ pc.em ∨ eb ≠ pc.eb ∨ etp ≠ pc.etp ∨ emp ≠ pc.emp ∨ ebp ≠ pc.ebp ∨ c8_4 ≠
        pc.c8_4 ∨ ctp ≠ pc.ctp ∨ cbp ≠ pc.cbp;
  }
```

See also sections 45 and 47.

This code is used in section 29.

43. To write our move method, we need arrays that give the action of moves on our various fields. For the edge group movement, the **kocsymm** class already provides this information, but it does not provide information on how the permutation of the constituent cubies changes. We need a move array that provides both pieces of information. The new coordinate requires nine bits to represent, and the S_4 index requires five bits to represent. We could use a three byte struct that would blow up to four bytes total for alignment, or we can use bit fields. We prefer bit fields; we code our own to make sure they fit in a short. We also need an array to manage the corner moves, with the same basic structure. We use file statics for these; no need to expose them.

```
⟨ Static data declarations for permcube 31 ⟩ +≡
  static unsigned short eperm_move[EDGEPERM][NMOVES];
  static int cperm_move[C8_4][NMOVES];
```

44. We instantiate those arrays here.

```
⟨ Static data instantiations 8 ⟩ +≡
  unsigned short permcube::eperm_move[EDGEPERM][NMOVES];
  int permcube::cperm_move[C8_4][NMOVES];
```

45. The move routine is declared here.

```
⟨ Methods for permcube 42 ⟩ +≡
  void move(int mv);
```

46. The move routine is pretty simple; for each group field, we calculate its new value, and extract the appropriate S_4 effect on the permutation of its elements from the low order five bits.

```
⟨ Method bodies 15 ⟩ +≡
  void permcube::move(int mv)
  {
#ifdef SAFETY_CHECKS
    if ((kocsymm::epsymm_expand[et] | kocsymm::epsymm_expand[em] |
         kocsymm::epsymm_expand[eb]) ≠ #fff)
      error ("! bad pc in move") ;
#endif
    int t = eperm_move[et][mv];
    et = t ≫ 5;
    etp = s4mul[etp][t & 31];
    t = eperm_move[em][mv];
    em = t ≫ 5;
    emp = s4mul[emp][t & 31];
    t = eperm_move[eb][mv];
    eb = t ≫ 5;
    ebp = s4mul[ebp][t & 31];
    t = cperm_move[c8_4][mv];
    c8_4 = t ≫ 10;
    ctp = s4mul[ctp][(t ≫ 5) & 31];
    cbp = s4mul[cbp][t & 31];
  }
```

47. In order to fill in these arrays, it's easiest to have a pair of routines that gets a permutation from a **cubepos**, and another that sets a permutation from a **cubepos**. Unlike **kocsymm**::*set_coset*, the *set_perm* routine will preserve the orientation, only affecting the cubie permutations. So if you call both *set_coset* and *set_perm*, make sure to call *set_coset* first and *set_perm* second.

We also provide routines to get only the corner information and only the edge information because sometimes that's all we need, and these routines can make a major difference in performance. We also provide a routine that gets just the up/down permutation and another that gets just the middle permutation for those specific cases where the position is guaranteed to be already in the Kociemba group. Similar routines exist for setting just the edge information and setting just the corner information.

```
⟨ Methods for permcube 42 ⟩ +≡
  void init_edge_from_cp(const cubepos &cp);
  void init_corner_from_cp(const cubepos &cp);
  permcube(const cubepos &cp);
  void set_edge_perm(cubepos &cp) const;
  void set_corner_perm(cubepos &cp) const;
  void set_perm(cubepos &cp) const;
```

48. The constructor from a basic cube simply iterates through the cubies, keeping track of which groups each cubie belongs to and the order that the cubies are seen in. We iterate backwards so the least significant cubie ends up in the low order bits. Edges first.

```
⟨ Method bodies 15 ⟩ +≡
  void permcube::init_edge_from_cp(const cubepos &cp)
  {
    et = em = eb = 0;
    etp = emp = ebp = 0;
    for (int i = 11; i ≥ 0; i--) {
      int perm = cubepos::edge_perm(cp.e[i]);
      if (perm & 4) {      /* middle layer */
        em |= 1 ≪ i;
        emp = 4 * emp + (perm & 3);
      }
      else if (perm & 8) {      /* bottom layer */
        eb |= 1 ≪ i;
        ebp = 4 * ebp + (perm & 3);
      }
      else {
        et |= 1 ≪ i;
        etp = 4 * etp + (perm & 3);
      }
    }
    et = kocsymm::epsymm_compress[et];
    em = kocsymm::epsymm_compress[em];
    eb = kocsymm::epsymm_compress[eb];
    etp = s4compress[etp];
    emp = s4compress[emp];
    ebp = s4compress[ebp];
  }
```

49. Corners next, plus the routine that puts the two together.

```
⟨ Method bodies 15 ⟩ +≡
  void permcube::init_corner_from_cp(const cubepos &cp)
  {
    c8_4 = 0;
    ctp = cbp = 0;
    for (int i = 7; i ≥ 0; i--) {
      int perm = cubepos::corner_perm(cp.c[i]);
      if (perm & 4) {      /* bottom layer */
        cbp = 4 * cbp + (perm & 3);
      }
      else {
        c8_4 |= 1 ≪ i;
        ctp = 4 * ctp + (perm & 3);
      }
    }
    c8_4 = c8_4_compact[c8_4];
    ctp = s4compress[ctp];
    cbp = s4compress[cbp];
  }
  permcube::permcube(const cubepos &cp)
  {
    init_edge_from_cp(cp);
    init_corner_from_cp(cp);
  }
```

50. The inverse routine is very similar, just in reverse. Edges first.

```
⟨ Method bodies 15 ⟩ +≡
  void permcube::set_edge_perm(cubepos &cp) const
  {
    int et_bits = kocsymm::epsymm_expand[et];
    int em_bits = kocsymm::epsymm_expand[em];
    int et_perm = s4expand[etp];
    int em_perm = s4expand[emp];
    int eb_perm = s4expand[ebp];
    for (int i = 0; i < 12; i++)
      if ((et_bits ≫ i) & 1) {      /* top layer */
        cp.e[i] = cubepos::edge_val((3 & et_perm), cubepos::edge_ori(cp.e[i]));
        et_perm ≫= 2;
      }
      else if ((em_bits ≫ i) & 1) {      /* middle layer */
        cp.e[i] = cubepos::edge_val((3 & em_perm) + 4, cubepos::edge_ori(cp.e[i]));
        em_perm ≫= 2;
      }
      else {      /* bottom layer */
        cp.e[i] = cubepos::edge_val((3 & eb_perm) + 8, cubepos::edge_ori(cp.e[i]));
        eb_perm ≫= 2;
      }
  }
```

51. Corners next, and we put it together.

```
⟨ Method bodies 15 ⟩ +≡
  void permcube::set_corner_perm(cubepos &cp) const
  {
    int c8_4_bits = c8_4_expand[c8_4];
    int ct_perm = s4expand[ctp];
    int cb_perm = s4expand[cbp];
    for (int i = 0; i < 8; i++)
      if ((c8_4_bits >> i) & 1) {      /* top layer */
        cp.c[i] = cubepos::corner_val((3 & ct_perm), cubepos::corner_ori(cp.c[i]));
        ct_perm >>= 2;
      }
      else {
        cp.c[i] = cubepos::corner_val((3 & cb_perm) + 4, cubepos::corner_ori(cp.c[i]));
        cb_perm >>= 2;
      }
  }
  void permcube::set_perm(cubepos &cp) const
  {
    set_edge_perm(cp);
    set_corner_perm(cp);
  }
```

52. Our base constructor is next. Everything can be set to zero, except for the *et* and *eb* bits, which must be set to values pulled from the *epsymm_expand* arrays. Since we are using file static objects for **kocsymm** that are defined in every compilation unit, we can assume that **kocsymm** and thus **permcube** is initialized at any point this constructor is called.

```
⟨ Method bodies 15 ⟩ +≡
  permcube::permcube( )
  {
    c8_4 = 0;
    ctp = cbp = 0;
    et = kocsymm::epsymm_compress[#f];
    em = 0;
    eb = kocsymm::epsymm_compress[#f00];
    etp = emp = ebp = 0;
  }
```

53. Now we are prepared to initialize the move arrays. We need to be careful to initialize the edge group variables to consistent values. We do the edges first.

```
⟨ Initialize permcube 34 ⟩ +≡
  cubepos cp, cp2;
  for (int i = 0; i < EDGEPERM; i++) {
    permcube pc;
    pc.em = i;
    int remaining_edges = ^fff - kocsymm::epsymm_expand[i];
    int mask = 0;
    int bitsseen = 0;
    while (bitsseen < 4) {
      if (remaining_edges & (mask + 1)) bitsseen++;
      mask = 2 * mask + 1;
    }
    pc.et = kocsymm::epsymm_compress[remaining_edges & mask];
    pc.eb = kocsymm::epsymm_compress[remaining_edges & ~mask];
    pc.set_perm(cp);
    for (int mv = 0; mv < NMOVES; mv++) {
      cp2 = cp;
      cp2.movepc(mv);
      permcube pc2(cp2);
      eperm_move[i][mv] = (pc2.em << 5) + pc2.emp;
    }
  }
```

54. The corner work is even easier; we follow the pattern we have established. In this case we need to calculate two permutation impacts, rather than just one. At the same time, we compute the edge up/down values.

```
⟨ Initialize permcube 34 ⟩ +≡
  for (int i = 0; i < C8_4; i++) {
    permcube pc;
    pc.c8_4 = i;
    pc.set_perm(cp);
    for (int mv = 0; mv < NMOVES; mv++) {
      cp2 = cp;
      cp2.movepc(mv);
      permcube pc2(cp2);
      cperm_move[i][mv] = (pc2.c8_4 << 10) + (pc2.ctp << 5) + pc2.cbp;
    }
  }
```

55. We terminate the **kocsymm** . *h* file here.

```
⟨ kocsymm.h 1 ⟩ +≡
#endif
```

56. Testing. We do some basic unit tests to verify functionality. First we do some basic tests; do our basic constructors generate the same thing as the identity cube?

```
⟨ Basic tests 56 ⟩ ≡
  {
    cubepos cpi;
    permcube pci(cpi);
    kocsymm kci(cpi);
    permcube pct;
    kocsymm kct;
    if (pct ≠ pci ∨ kct ≠ kci)
      error ("! problem with default constructors") ;
    if (permcube::c12_8[pc.et] ≠ 0)
      error ("! bad mapping in 12->8") ;
  }
```

This code is used in section 62.

57. Next we test conversions. If we start with a random **cubepos**, can we convert it to a pair of **kocsymm** and **permcube** structures, and then back again, with no loss of data? Also, does the edge permutation coordinate in the **kocsymm** match the *em* field of the **permcube** ;

```
⟨ Test conversions back and forth 57 ⟩ ≡
  for (int i = 0; i < 100000; i++) {
    cp.randomize( );
    kocsymm kc(cp);
    permcube pc(cp);
    if (kc.epsymm ≠ pc.em)
      error ("! mismatch in edge middle occupancy") ;
    kc.set_coset(cp2);
    pc.set_perm(cp2);
    if (cp ≠ cp2)
      error ("! mismatch in conversion and back") ;
  }
```

This code is used in section 62.

58. Next we test the move routines. Do we get the same results when we use **permcube** and **kocsymm** as we do when we use **cubepos**?

```
⟨ Test move routines 58 ⟩ ≡
  for (int i = 0; i < 1000; i++) {
    cp.randomize( );
    kocsymm kc(cp);
    permcube pc(cp);
    int mv = random_move_ext( );
    cp.movepc(mv);
    cp2 = cp;
    kc.move(mv);
    pc.move(mv);
    kc.set_coset(cp2);
    pc.set_perm(cp2);
    if (cp ≠ cp2)
      error ("! mismatch in move test") ;
  }
```

This code is used in section 62.

59. The next thing we test is canonicalization in **kocsymm**.

```
⟨ Test canonicalization 59 ⟩ ≡
  for (int i = 0; i < 1000; i++) {
    cp.randomize( );
    kocsymm kc(cp);
    for (int m = 1; m < KOCSYMM; m++) {
      cp.remap_into(m, cp2);
      kocsymm kc2(cp2);
      if (kc2 < kc) kc = kc2;
    }
    kocsymm kc3(cp);
    kc3.canon_into(kc2);
    if (kc2 ≠ kc)
      error ("! canonicalization failuree") ;
  }
```

This code is used in section 62.

60. Finally, we count how many canonical cosets of H there are. This takes a second or two.

```
⟨ Count cosets 60 ⟩ ≡
  int s = 0;
  for (int c = 0; c < CORNERSYMM; c++) {
    int bits = kocsymm::cornersymm[c].minbits;
    if (bits ≡ 1) {
      s += EDGEOSYMM * EDGEPERM;
    }
    else if (bits & 1) {
      for (int eo = 0; eo < EDGEOSYMM; eo++)
        for (int ep = 0; ep < EDGEPERM; ep++) {
          kocsymm kc(c, eo, ep);
          kc.canon_into(kc2);
          if (kc ≡ kc2) s++;
        }
    }
  }
  cout ≪ "Final sum is " ≪ s ≪ endl;
  if (s ≠ 138639780)
    error ("! bad total coset calculation") ;
```

This code is used in section 62.

61. We do some move timing tests. We have to make sure to use the results, or the compiler might eliminate the code. Note that these timing tests are for random moves, which may not be indicative of actual performance in a real program; we only include these timings out of general interest. In general we expect the **kocsymm** moves to be fastest, followed by **permcube**, and the two routines for **cubepos** will probably bring up the rare. In any case, we expect any of the move routines to take only a few dozen nanoseconds.

```
⟨ Move timing tests 61 ⟩ ≡
  int mvs[10000];
  for (int i = 0; i < 10000; i++) mvs[i] = random_move( );
  duration( );
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) kc.move(mvs[j]);
  cout ≪ "Moving 100M kc took " ≪ duration( ) ≪ endl;
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) pc.move(mvs[j]);
  cout ≪ "Moving 100M pc took " ≪ duration( ) ≪ endl;
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) cp.move(mvs[j]);
  cout ≪ "Moving 100M cp (move) took " ≪ duration( ) ≪ endl;
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) cp.movepc(mvs[j]);
  cout ≪ "Moving 100M cp (movepc) took " ≪ duration( ) ≪ endl;
  if (cp < cp2 ∧ pc < pc2 ∧ kc < kc2) cout ≪ "(Ignore this message.)" ≪ endl;
```

This code is used in section 62.

62. We put all the pieces together in our main test routine.

```
⟨ kocsymm_test.cpp 62 ⟩ ≡
#include "kocsymm.h"
#include <iostream>
  using namespace std;
  int main(int argc, char *argv[])
  {
    if (lrand48() ≡ 0) srand48(getpid() + time(0));
    kocsymm kc, kc2;
    permcube pc, pc2;
    cubepos cp, cp2;
    ⟨ Basic tests 56 ⟩
    ⟨ Test conversions back and forth 57 ⟩
    ⟨ Test move routines 58 ⟩
    ⟨ Test canonicalization 59 ⟩
    ⟨ Count cosets 60 ⟩
    ⟨ Move timing tests 61 ⟩
  }
```

KOCSYMM