1. Introduction.

FIST

Emphatic Message Generator

by John Walker

A year or two after emigrating, she happened to be in Paris on the anniversary of the Russian invasion of her country (Czechoslovakia). A protest march had been scheduled, and she felt driven to take part. Fists raised high, the young Frenchmen shouted out slogans condemning Soviet imperialism. She liked the slogans, but to her surprise she found herself unable to shout along with them. She lasted only a few minutes in the parade.

When she told her French friends about it, they were amazed. "You mean you don't want to fight the occupation of your country?" She would have liked to tell them that behind Communism, Fascism, behind all occupations and invasions lurks a more basic, pervasive evil and that the image of that evil was a parade of people marching with raised fists and shouting identical syllables in unison. But she knew she would never be able to make them understand. Embarrassed, she changed the subject.

— Milan Kundera, *The Unbearable Lightness of Being*

```
#define PRODUCT "fist"
#define VERSION "4.0"
#define REVDATE "2001-11-24"
```

2. Command line.

`FIST` is invoked with a command line as follows:

`fist` *options*

where *options* specify processing modes as defined below. Input is read from standard input until end of file is encountered and output is sent to standard output. All I/O is done serially and the program may be used as part of a pipeline.

3. Options.

Options are specified on the command line, preceded by a hyphen, "`-`". Arguments for options which take them must immediately follow the option letter. Arguments without options may not be aggregated.

`-c`
: Don't print the fist—just the block letter message.

`-f` *pat*
: Fill the fist with the text pattern *pat* (default "`*`").

`-l`
: Left handed fist.

`-m` *text*
: Use *text* as message text. You can specify multiple `-m` options for multiple line messages. If the `-n` option is not specified, lines from standard input will be printed after lines given by `-m` options.

`-n`
: Do not read message from standard input.

`-s` *factor*
: Scale the fist by the given percentage $25 \leq factor \leq 100$.

`-u`
: Print how to call information.

4. Main program.
The main program has a simple straight-through one pass structure. The section names indicate the phases of processing.

```
⟨ Main program 4 ⟩ ≡
  int main(int argc, char *argv[])
  {
    ⟨ Declare local variables 5 ⟩;
    ⟨ Process command line options 17 ⟩;
    ⟨ Generate and print the fist 6 ⟩;
    ⟨ Process text specified by command line M options 13 ⟩;
    ⟨ Read input lines and write as block letters below the fist 14 ⟩;
    return 0;
  }
```

This code is used in section 21.

5. The following variables are defined in the context of the main program. No dynamically allocated storage is used.

```
⟨ Declare local variables 5 ⟩ ≡
  int i, j, l, xstart, ystart, xend, yend, fillen, prfist = TRUE, readin = TRUE, mopt = FALSE,
      left = FALSE, scale = 100, sPLEN, sXLEN;
  char *cp, opt, *fillpat = FILL;
  char ibfr[200];
  char page[PLEN][XLEN + 2];
  char llen[PLEN];
```

This code is used in section 4.

6. Generate and print the fist.

```
⟨ Generate and print the fist 6 ⟩ ≡
  if (prfist) {
    ⟨ Generate the fist in memory 7 ⟩;
    ⟨ Print the fist 12 ⟩;
  }
```

This code is used in section 4.

7. The fist is generated in three phases. First we initialise the in memory arrays, then scan the table of rectangles to be filled, and finally add line terminators after the rightmost character filled in each line.

```
⟨ Generate the fist in memory 7 ⟩ ≡
  ⟨ Compute scaled image size 8 ⟩;
  ⟨ Initialise fist and line length arrays 9 ⟩;
  ⟨ Fill in rectangles comprising the fist 10 ⟩;
  ⟨ Place C string terminators at the end of each line 11 ⟩;
```

This code is used in section 6.

8. The original fist image was encoded in 1969, the very height of the mainframe era, when real programmers punched cards and line printers used wide paper with 120 characters or more per line (132 in the case of UNIVAC printers: 22 36-bit words of 6 FIELDATA characters each). The image was sized to fit a wider page than is common on modern-day window displays and printers—it requires a line length of 96 character or more to avoid truncation or wrapping around. To preserve the integrity of the original 30+ year old image while accommodating contemporary community standards, the `-s` option allows you specify a scale factor between 25% and 100% (the latter the default if the option is omitted) which applies to both the width and height of the fist image. Here we take the scale factor scanned in ⟨ Process command line options 17 ⟩ or the default and compute the actual width and height of the image to generate. We define a *Scale* macro which applies this scale factor to co-ordinate pairs extracted from the table of filled rectangles in ⟨ Fill in rectangles comprising the fist 10 ⟩ below.

```
#define Scale(x, y) x = (x * scale)/100;
        y = (y * scale)/100;
```

⟨ Compute scaled image size 8 ⟩ ≡

```
sPLEN = (PLEN * scale)/100;
sXLEN = (XLEN * scale)/100;
```

This code is used in section 7.

9. The fist image is built in the *page* array. The rectangle table, *fistab*, only specifies the areas to be filled so we need to initialise the array to all blanks so areas not filled will print properly. We also clear the *llen* array to all zeroes. This array keeps track of the rightmost character filled on each line and permits ⟨ Place C string terminators at the end of each line 11 ⟩ to insert C string terminators so no trailing spaces are printed.

⟨ Initialise fist and line length arrays 9 ⟩ ≡

```
memset(llen, 0, sizeof llen);
for (j = 0; j < sPLEN; j++) {
  memset(page[j], '␣', sPLEN);
}
```

This code is used in section 7.

10. The fist is defined in the *fistab* array as corners of rectangles in the *page* array to be filled with *fillpat*. The last rectangle in the table is indicated by its first Y co-ordinate being zero. If the `-l` option is specified, we mirror the image in the X axis to change it into a left handed fist.

```
⟨ Fill in rectangles comprising the fist 10 ⟩ ≡
  fillen = strlen(fillpat);
  l = 0;
  while (TRUE) {
    xstart = fistab[l++];
    if (¬(ystart = fistab[l++])) {
      break;
    }
    xend = fistab[l++];
    yend = fistab[l++];
    Scale(xstart, ystart);
    Scale(xend, yend);
    if (left) {
      int tx = sXLEN − xstart;
      xstart = sXLEN − xend;
      xend = tx;
    }
    for (i = ystart; i ≤ yend; i++) {
      for (j = xstart; j ≤ xend; j++) {
        page[i][j] = fillpat[j % fillen];
      }
      if (xend > llen[i]) {
        llen[i] = xend;
      }
    }
  }
```

This code is cited in section 8.

This code is used in section 7.

11. As the rectangles from *fistab* were filled in the *page* array, the rightmost character in each line is kept track of in *llen*. Once all of the rectangles have been filled, we riffle through the lines and place a C line terminator at the end of each line.

```
⟨ Place C string terminators at the end of each line 11 ⟩ ≡
  for (i = 0; i < sPLEN; i++) {
    page[i][llen[i] + 1] = EOS;
  }
```

This code is cited in section 9.

This code is used in section 7.

12. With the fist now built in the *page* array, we need only iterate over the lines, printing each in turn.

```
⟨ Print the fist 12 ⟩ ≡
  for (i = 0; i < sPLEN; i++) {
    puts(page[i]);
  }
```

This code is used in section 6.

13. If one or more "`-m`" options were specified on the command line, ⟨ Process command line options 17 ⟩ sets the *mopt* flag to remind us to re-scan the command line and process them here. Arguments of the `-m` options are passed successively to *printmsg* to print them in the order they appear on the command line.

```
⟨ Process text specified by command line M options 13 ⟩ ≡
  if (mopt) {
    for (i = 0; i < argc; i++) {
      cp = argv[i];
      if ((cp[0] ≡ '-') ∧ ((cp[1] ≡ 'm') ∨ cp[1] ≡ 'M')) {
        printmsg(cp + 2);
      }
    }
  }
```

This code is used in section 4.

14. If the "`-n`" option is not specified, after processing any text from command line `-m` options, lines are read from standard input and printed below the fist. Lines from standard input are processed until end of file is encountered.

```
⟨ Read input lines and write as block letters below the fist 14 ⟩ ≡
  if (readin) {
    while (fgets(ibfr, (sizeof ibfr) − 1, stdin)) {
      ibfr[strlen(ibfr) − 1] = 0;
      printmsg(ibfr);
    }
  }
```

This code is used in section 4.

15. Lines of text below the fist are printed by the *printmsg* function. A maximum of 16 characters are printed from each line; any additional characters are truncated.

⟨ Print message in block characters 15 ⟩ ≡

```
static void printmsg(char *ibfr)
{
  int c, i, j, l;
  char oline[134];
  l = strlen(ibfr);
  if (l > 16) {
    l = 16;
  }
  puts("\n");
  for (i = 0; i < 8; i++) {
    oline[0] = EOS;
    for (j = 0; j < l; j++) {
      if (j > 0) {
        strcat(oline, "␣␣");
      }
      c = ibfr[j];
      if (islower(c)) {
        c = toupper(c);
      }
      if ((c < '␣') ∨ (c > '_')) {
        c = '␣';
      }
      strcat(oline, bitstring(c, i));
    }
    puts(oline);
  }
}
```

This code is used in section 24.

16. The *bitstring* function decodes the block character font defined in *chartab* below. The function is called with the ASCII character code c and the row index i and returns a pointer to a **static** string (yikes—non-reentrant!) containing a zero-terminated string of 6 characters containing the row of the character. The representation of the character is always made up of spaces and the character itself.

```
⟨ Decode block character font table 16 ⟩ ≡
  static char *bitstring(int c, int i)
  {
    static char r[8];
    int bits, b, n;
    bits = (chartab[(i ≫ 1) + ((c − ' ') ≪ 2)] ≫ ((i & 1) ? 0 : 8)) & #FF;
    b = 32;
    n = 0;
    while (b) {
      r[n++] = (bits & b) ? c : ' ';
      b ≫= 1;
    }
    r[n] = EOS;
    return r;
  }
```

This code is used in section 24.

17. We scan the command line options with our own old-fashioned traditional option cruncher. There are so few options and they're so trivial to parse it's hard to justify going to all the trouble of using *getopt*. Preserving the tradition of a program which, among its myriad incarnations, ran on MS-DOS, options are accepted in either upper or lower case.

```
⟨ Process command line options 17 ⟩ ≡
  for (i = 0; i < argc; i++) {
    cp = argv[i];
    if (*cp ≡ '-') {
      opt = *(++cp);
      if (islower(opt)) opt = toupper(opt);
      switch (opt) {
      case 'C':     /* -c      Don't print the fist */
        prfist = FALSE;
        break;
      case 'F':     /* -fpat     Fill fist with pat pattern */
        if (cp[1] ≠ 0) {
          fillpat = cp + 1;
        }
        break;
      case 'L':     /* -l      Left handed fist */
        left = TRUE;
        break;
      case 'M':     /* -mtext     Specify message on command line */
        mopt = TRUE;     /* Just remember we've seen an -m—processing occurs on the second pass over
            the options after the fist is printed. */
        break;
      case 'N':     /* -n      Don't read standard input */
        readin = FALSE;
        break;
      case 'S':     /* -sfactor     Scale fist by factor percent, 25 ≤factor≤ 100 */
        scale = atoi(cp + 1);
        if (scale < 25) {
          scale = 25;
        }
        else if (scale > 100) {
          scale = 100;
        }
        break;
      case '?': case 'U':     /* -u      Print how to call information */
        ⟨ Print how to call information 18 ⟩;
        return 0;
      }
    }
  }
```

This code is cited in sections 8 and 13.

This code is used in section 4.

18. Print the how to call information when the "`-u`" option is specified. Such information was traditionally sent to standard error, but contemporary GNU software prints to standard output, as we do here.

⟨ Print how to call information 18 ⟩ ≡

```
printf("FIST␣--␣␣Emphatic␣message␣program.␣␣Call\n");
printf("␣␣␣␣␣␣␣␣␣␣with␣fist␣[options]␣<input␣>output\n");
printf("\n");
printf("␣␣␣␣␣␣␣␣␣␣Options:␣␣␣-C␣␣␣␣␣␣␣␣Cool␣it,␣baby␣(no␣fist)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-Fpat␣␣␣␣Fill␣with␣pattern␣pat\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-L␣␣␣␣␣␣␣␣Left␣handed␣fist\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-Mtext␣␣␣␣Use␣text␣as␣message␣below␣fist␣"
    "(multiple␣-M␣options␣OK)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-N␣␣␣␣␣␣␣␣Don't␣read␣message␣from␣standard␣input\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-Sfactor␣Scale␣fist␣by␣25%%␣<=␣factor␣<=␣100%%\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-U,␣-?␣␣␣␣Print␣this␣message\n");
printf("\n");
printf("␣␣␣␣␣␣by␣John␣Walker␣(http://www.fourmilab.ch/)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣Release␣%s␣--␣␣%s\n", VERSION, REVDATE);
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(P)␣All␣Rights␣Reversed\n");
```

This code is used in section 17.

19. Block character font table.

The character table is stored as an array of **short** integers indexed by the character codes for ASCII characters from #20 "␣" through #5F "_". *Characters outside this range are not defined!* You should map lower case letters to upper case and replace all unavailable characters with a replacement such as blank or question mark. The font is defined in a 5×7 matrix with each row of 5 characters bit-coded in the least significant bits of a byte. Successive rows are stored in the four **short**s for each character, least significant byte first. The curious encoding and the fact that we store these in **short**s at all is, like everything else in this program, an artefact of history.

The following block character font was created in Cleveland in about 1968. I have forgotten the name of the creator—if you recall, please let me know. The font table was originally made for the six bit FIELDATA code used by the UNIVAC 1108; it has been reshuffled into ASCII order and a few characters present in ASCII but not FIELDATA added.

```
⟨ Global variables 19 ⟩ ≡
  static short chartab[] = {
  0, 0, 0, 0      /* ␣ */
  , 2056, 2056, 8, 2048      /* ! */
  , 5140, 5120, 0, 0      /* " */
  , 20, 15892, 15892, 0      /* # */
  , 2110, 10302, 2622, 2048      /* $ */
  , 2, 9224, 4640, 0      /* % */
  , 7202, 10264, 10274, 7168      /* & */
  , 2056, 0, 0, 0      /* ' */
  , 2064, 8224, 8208, 2048      /* ( */
  , 4104, 1028, 1032, 4096      /* ) */
  , 42, 7230, 7210, 0      /* * */
  , 8, 2110, 2056, 0      /* + */
  , 0, 0, 6152, 4096      /* , */
  , 0, 62, 0, 0      /* - */
  , 0, 0, 24, 6144      /* . */
  , 2, 1032, 4128, 0      /* / */
  , 7202, 8738, 8738, 7168      /* 0 */
  , 2072, 2056, 2056, 7168      /* 1 */
  , 7202, 1032, 4128, 15872      /* 2 */
  , 7202, 540, 546, 7168      /* 3 */
  , 1036, 5182, 1028, 1024      /* 4 */
  , 15904, 8252, 546, 7168      /* 5 */
  , 7202, 8252, 8738, 7168      /* 6 */
  , 15874, 1032, 4128, 8192      /* 7 */
  , 7202, 8732, 8738, 7168      /* 8 */
  , 7202, 8734, 546, 7168      /* 9 */
  , 0, 2048, 2048, 0      /* : */
  , 24, 6144, 6152, 4096      /* ; */
  , 516, 2064, 2052, 512      /* < */
  , 0, 15872, 15872, 0      /* = */
  , 4104, 1026, 1032, 4096      /* > */
  , 7202, 1032, 2048, 2048      /* ? */
  , 7202, 10798, 8226, 7168      /* @ */
  , 2068, 8738, 15906, 8704      /* A */
  , 15394, 8764, 8738, 15360      /* B */
  , 7202, 8224, 8226, 7168      /* C */
  , 15394, 8738, 8738, 15360      /* D */
  , 15904, 8248, 8224, 15872      /* E */
```

```
  , 15904, 8248, 8224, 8192      /* F */
  , 7202, 8238, 8738, 7168       /* G */
  , 8738, 8766, 8738, 8704       /* H */
  , 7176, 2056, 2056, 7168       /* I */
  , 514, 514, 8738, 7168       /* J */
  , 8740, 10288, 10276, 8704      /* K */
  , 8224, 8224, 8224, 15872      /* L */
  , 8758, 10786, 8738, 8704      /* M */
  , 8754, 8746, 8742, 8704       /* N */
  , 7202, 8738, 8738, 7168       /* O */
  , 15394, 8764, 8224, 8192      /* P */
  , 7202, 8738, 10790, 7680      /* Q */
  , 15394, 8764, 10276, 8704      /* R */
  , 7202, 8220, 546, 7168      /* S */
  , 15880, 2056, 2056, 2048      /* T */
  , 8738, 8738, 8738, 7168       /* U */
  , 8738, 8738, 8724, 2048       /* V */
  , 8738, 8738, 10806, 8704      /* W */
  , 8738, 5128, 5154, 8704       /* X */
  , 8738, 5128, 2056, 2048       /* Y */
  , 15874, 1032, 4128, 15872      /* Z */
  , 7184, 4112, 4112, 7168       /* [ */
  , 32, 4104, 1026, 0      /* \ */
  , 7172, 1028, 1028, 7168       /* ] */
  , 8, 5154, 0, 0      /* ^ */
  , 0, 0, 0, 15872      /* _ */
  };
```

See also section 20.

This code is used in section 21.

20. Fist image table.

The fist image is defined by *fistab*, an array of **char** consisting of quadruples representing co-ordinates of the upper left and lower right corners of a square to be filled with the *fillpat* sequence. The list of squares is terminated by a quadruple of all zeroes (actually, scanning of the table is terminated by a zero as the second item of the quadruple—the others are not tested). This table defines a right handed fist; when the `-l` option is specified, X co-ordinates are mirrored to obtain a left hand.

⟨ Global variables 19 ⟩ +≡

```
static char fistab[] = {
26, 36, 64, 102, 40, 3, 57, 14, 41, 2, 56, 2, 44, 1, 54, 1, 58, 8, 58, 12, 43, 15, 56, 22, 43, 23, 49, 23, 56, 14, 57, 20, 56, 21,
    56, 21, 41, 15, 42, 17, 42, 18, 42, 21, 65, 79, 65, 102, 49, 103, 65, 107, 35, 103, 48, 106, 23, 103, 34, 105, 14, 103,
    22, 104, 10, 103, 21, 103, 13, 92, 25, 102, 12, 95, 24, 102, 11, 98, 23, 102, 18, 78, 25, 91, 17, 82, 17, 91, 16, 85,
    16, 91, 15, 87, 15, 91, 14, 89, 14, 91, 22, 67, 25, 77, 21, 70, 21, 77, 20, 72, 20, 77, 19, 75, 19, 77, 10, 31, 21, 43,
    22, 32, 25, 56, 23, 57, 25, 58, 24, 59, 24, 59, 15, 44, 25, 48, 16, 49, 25, 49, 17, 50, 25, 50, 19, 51, 25, 51, 21, 52,
    25, 52, 22, 53, 25, 53, 23, 54, 25, 54, 24, 56, 25, 56, 11, 44, 14, 44, 12, 45, 14, 45, 13, 46, 14, 46, 14, 46, 15, 46, 8,
    31, 9, 42, 7, 32, 7, 41, 6, 33, 6, 39, 66, 25, 84, 50, 65, 34, 65, 67, 59, 35, 64, 35, 62, 34, 64, 34, 69, 22, 80, 24, 81,
    23, 82, 23, 81, 24, 83, 24, 47, 25, 65, 32, 48, 33, 61, 33, 46, 26, 46, 31, 45, 28, 45, 29, 85, 34, 93, 43, 85, 44, 92,
    44, 85, 45, 91, 45, 85, 46, 90, 46, 85, 47, 89, 47, 85, 48, 87, 48, 85, 49, 86, 49, 94, 35, 94, 42, 95, 37, 95, 40, 96,
    38, 96, 39, 85, 31, 90, 33, 85, 30, 89, 30, 88, 29, 88, 29, 91, 32, 91, 33, 92, 33, 92, 33, 58, 28, 87, 29, 85, 27, 86,
    27, 85, 26, 85, 26, 65, 51, 74, 56, 65, 57, 73, 57, 65, 58, 72, 58, 65, 59, 71, 59, 65, 60, 70, 60, 65, 61, 68, 61, 65,
    62, 67, 62, 65, 63, 66, 63, 65, 64, 65, 64, 75, 51, 83, 51, 75, 52, 81, 52, 75, 53, 79, 53, 75, 54, 77, 54, 75, 55, 75,
    55, 22, 35, 43, 35, 22, 34, 38, 34, 22, 33, 32, 33, 22, 32, 24, 32, 61, 8, 78, 16, 62, 7, 78, 7, 62, 6, 77, 6, 63, 5, 76, 5,
    61, 17, 70, 20, 71, 17, 76, 18, 71, 19, 74, 19, 26, 4, 35, 29, 36, 5, 36, 29, 37, 12, 37, 29, 38, 16, 38, 29, 39, 20, 39,
    27, 40, 26, 40, 27, 27, 30, 38, 30, 29, 31, 35, 31, 20, 4, 25, 14, 24, 3, 33, 3, 19, 5, 19, 12, 18, 6, 18, 10, 22, 15, 25,
    20, 21, 15, 21, 18, 20, 15, 20, 16, 25, 21, 25, 29, 24, 21, 24, 25, 23, 21, 23, 22, 59, 24, 70, 24, 64, 23, 70, 23, 11,
    22, 20, 27, 21, 24, 21, 27, 16, 28, 19, 28, 11, 30, 18, 30, 5, 11, 15, 21, 6, 22, 10, 22, 7, 23, 10, 23, 8, 24, 10, 24, 9,
    25, 10, 25, 10, 26, 10, 26, 2, 12, 4, 18, 1, 13, 1, 17, 2, 19, 4, 19, 3, 20, 4, 20, 4, 11, 4, 11, 7, 10, 15, 10, 12, 9, 13, 9,
    16, 14, 18, 22, 16, 12, 17, 13, 19, 18, 19, 21, 20, 20, 20, 21, 25, 59, 25, 66, 24, 65, 24, 66, 23, 66, 23, 66, 0, 0, 0, 0
};
```

21. Putting the pieces together.

Now we collect together all the various pieces into the complete program, organised so the compiler can figure out what we're doing.

⟨ Preprocessor definitions ⟩
⟨ System include files 23 ⟩
⟨ Global variables 19 ⟩
⟨ Global functions 24 ⟩
⟨ Main program 4 ⟩

22. We define the following constants at the global level.

```
#define FALSE 0     /* What is truth? */
#define TRUE 1
#define EOS '\0'    /* C end of string marker */
#define XLEN 96     /* Maximum line length */
#define PLEN 108    /* Page length */
#define FILL "*"    /* Default fill pattern */
```

23. System include files.

The following include files provide access to system and library components. These files are conditionally included based on `config`'s determination of which were present on the system where we're building.

⟨ System include files 23 ⟩ ≡

```
#include "config.h"
#include <stdio.h>
#include <ctype.h>
#ifdef HAVE_STDLIB_H
#include <stdlib.h>
#endif
#ifdef HAVE_STRING_H
#include <string.h>
#endif
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
```

This code is used in section 21.

24. To avoid ugly forward function declarations, we collect the **static** utility functions at the top of the C file so the compiler sees them before they're referenced in the main program.

⟨ Global functions 24 ⟩ ≡

```
⟨ Decode block character font table 16 ⟩
⟨ Print message in block characters 15 ⟩
```

This code is used in section 21.

25. Release history.

Release 1: September 1969

Originally written (on punch cards) in Case ALGOL for the UNIVAC 1108. The fist was digitised and encoded as rectangles by hand and has not changed in any subsequent edition. The block character font was adapted from a banner making program whose details and author I have since forgotten.

Release 2: August 1981

Rewritten in Marinchip QBASIC and released on the Marinchip giveaway software disc.

Release 3: October 1985

Rewritten in K&R C.

Release 4: November 2001

Converted to ANSI C, re-organised as a `CWEB` literate program. Added the `-l`, `-m`, `-n`, and `-s` options and the ability to specify a multi-character fill pattern with the `-f` option.

26. Bugs.

- The block character font is a limited subset of ASCII containing only upper case letters, numbers, and punctuation with character codes between hexadecimal #20 and #5F. The font was originally created in the late 1960's in UNIVAC 1108 six bit FIELDATA code, and re-shuffled into ASCII order when the first port was made to an ASCII machine in 1981. If you'd like to add lower case letters, ISO codes, or full Unicode, go right ahead.

- Scale factors smaller than about 60 on the `-s` option produce infelicitous results: the fist looks like it's wearing a mitten.

- `FIST` will not work on machines which do not use the ASCII character code (for example EBCDIC machines). You'll need to shuffle the font table or translate character codes to ASCII before you index it. I don't have such a machine, so I'm not going to include code I can't test.

- You can't aggregate options, separate options from their arguments with a blank, or other cool *getopt* features because the program doesn't use *getopt* in order to preserve its retro look.

- Over the last thirty years numerous people have suggested the program might be enhanced by adding options to raise two fingers ("peace") or only one finger (well, you know). Please send me the code if you make this decades-long dream a reality.

27. Index. The following is a cross-reference table for `fist`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

FIST