```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: January 23, 2003
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th
 * http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Computer Systems (2301274) classnote.
 * Description: This sample table lookup construction illustrates
 *              a straightforward direct use of predefined constants
 *              to build an instruction set.
 * Note: the function prototypes are placed just below the structure
 * definition to ensure that 'template' is defined before used in
 * 'build_tree' module prototype.
 *
 * History of Modifications:
 *    (1) January 29, 2003
 *        -add getopt call to parse the command line for any options
 *         specified by the user.
 *        -add error code for higher level invocation (from shell script).
 *        -migrate mainstream processing down the calling hierarchy,
 *         leaving the main program with only necessary work.
 *        A number of parameters were incorporated in several functions
 *        as a result of the above modifications, namely, fill_struct,
 *        special_case, main_loop, and out_put(newly added).
 */

#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <sys/types.h>

#define      EQ(a, b)            (strcmp(a, b) == 0)
#define      NE(a, b)            (strcmp(a, b) != 0)
#define      Null                ""
#define      Succeeded           1
#define      Failed              0
#define      Yes                 1
#define      No                  0
#define      Size1               4
#define      Size2               5

#define      rmode               "r"
#define      wmode               "w"
#define      Remove_File         "rm -f"
#define      Fname_size          40

/*
 * return code to OS
 */
#define      Normal              0
#define      Error_option        1
#define      Missing_file        2
#define      Openf_error         3
#define      Out_of_mem          4
```

```
/*
 * Globals
 */
char   *reg_instr[] =
{
       "AND", "ADD", "LDA", "STA", "BUN", "BSA", "ISZ",
       Null
};

char   *irreg_instr[]      =
{
       "CLA", "CLE", "CMA", "CME", "CIR", "CIL", "INC",
       "SPA", "SNA", "SZA", "SZE", "HLT",
       "INP", "OUT", "SKI", "SKO", "ION", "IOF",
       Null
};

char   *bin_code[]  =
{
       "7800", "7400", "7200", "7100", "7080", "7040", "7020",
       "7010", "7008", "7004", "7002", "7001",
       "F800", "F400", "F200", "F100", "F080", "F040",
       Null
};

char   *macros[]    =
{
       "ORG", "END", "DEC", "HEX",
       Null
};

struct instructions
{
       char   opcode[Size1];
       char   bin_equiv[Size2];
};
typedef      struct instructions  template;

/*
 * just to demonstrate debug mode, as well as other options, by means
 * of the global variables.
 */
int          debug = No;
int          keepf = No;
extern int   optind;
extern char  *optarg;
```

```
/*
 * function prototypes
 */
void   *malloc(size_t);
void   free(void *);
void   special_case(char *, FILE *);
void   free_all(void);
void   out_put(template *, FILE *);
int    fill_struct(char *, char *, FILE *);
int    build_tree(template *);
int    main_loop(char *, char *);

/*
 * Main return 0 as normal exit status to OS and 1 otherwise.
 * 'count' keeps the number of elements of 'reg_instr'.
 * A full option invocation of this program can take one of
 * the following combinations:
 *   (1) Program_name  -k input_file -o output_file -d
 *   (2) Program_name  -d -k input_file -o output_file
 *   (3) Program_name  -o output_file -k input_file -d
 *   (4) Program_name  input_file -d -o output_file -k
 *   (5) Program_name  input_file -d -k -o output_file
 *   (6) Program_name  -d -k -o output_file input_file
 *   and so on...
 * Note that the output filename must follow the option 'o'
 * immediately as its argument.
 */
int
main(int argc, char *argv[])
{
      int    return_OS;
      int    c, errflg = 0;
      char   ifile[Fname_size];
      char   ofile[Fname_size];

      /*
       * Initialize ifile and ofile to NULL, then scan for
       * options 'd', 'o', and 'k' from the command line.
       */
      strcpy(ifile, Null);
      strcpy(ofile, Null);
      while ((c = getopt(argc, argv, "do:k")) != EOF)
      {
            switch (c)
            {
                  case 'd':
                        debug = Yes;
                        break;
                  case 'o':
                        strcpy(ofile, optarg);
                        break;
                  case 'k':
                        keepf = Yes;
                        break;
                  default:
                        errflg++;
            }
      }
```

```
	if (argc == 1 || errflg || argc == optind)
	{
		if (argc == optind)
		{
			fprintf(stderr, "Missing input filename\n");
			return_OS = Missing_file;
		}
		else
			return_OS = Error_option;
		fprintf(stderr, "Usage: %s  input_filename   ", argv[0]);
		fprintf(stderr, "[-o output_filename]  [-d]  [-k]\n");
	}
	else
	{
		for (; optind < argc; optind++)
		{
			/*
			 * Pick up only the first remaining argument to
			 * be the input filename.  The rest is discarded.
			 */
			strcpy(ifile, argv[optind]);
			break;
		}
		return_OS = main_loop(ifile, ofile);
		free_all();
	}
	return return_OS;
}

/*
 * main_loop receives the input and output filenames from main program.
 * input:  input and output filenames
 * output: error condition to inform 'main' of the execution outcome.
 */
int
main_loop(char *infile, char *outfile)
{
	FILE	*fp1, *fp2;
	int	return_code = Normal;
	int	indx, icnt;
	int	count, yes_pass;
	char	dummy[Size2];
	char	tmp[Fname_size];

	/*
	 * the 'infile' and 'outfile' are for demonstration purpose only.
	 */
	if ((fp1 = fopen(infile, rmode))  == NULL)
		return Openf_error;
	if (EQ(outfile, Null))
		fp2 = stdout;
	else
	{
		if ((fp2 = fopen(outfile, wmode)) == NULL)
		{
			fclose(fp1);
			return Openf_error;
		}
	}
```

```
	count = (int) (sizeof(reg_instr) / sizeof(char *));
	/*
	 * 'reg_instr' is called twice, hence the repeat.
	 */
	for (yes_pass = Yes, icnt = indx = 0; indx < count; indx++, icnt++)
	{
		if (EQ(reg_instr[indx], Null))
		{
			if (yes_pass == Yes)
			{
				yes_pass = No;
				indx = -1;
			}
			else
				break;
		}
		else
		{
			sprintf(dummy, "%-*X", Size2, icnt);
			if (fill_struct(reg_instr[indx], dummy, fp2) == Failed)
				return_code = Out_of_mem;
		}
	}

	/*
	 * special instructions are called only once.
	 */
	for (indx = 0; NE(irreg_instr[indx], Null); indx++)
	{
		sprintf(dummy, "%-*s", Size2, bin_code[indx]);
		if (fill_struct(irreg_instr[indx], dummy, fp2) == Failed)
			return_code = Out_of_mem;
	}
	for (indx = 0; NE(macros[indx], Null); indx++)
		special_case(macros[indx], fp2);

	free_all();
	fclose(fp1);
	fclose(fp2);

	if (keepf == No && NE(outfile, Null))
	{
		sprintf(tmp, "%s %s", Remove_File, outfile);
		system(tmp);
	}

	return return_code;
}
```

```
/*
 * Fill the template with mnemonic and binary equivalent to form
 * partial assembly instruction.  In case of 'irreg_instr',
 * concatenate the location counter to complete the instruction.
 * If build_tree fails, the function will free current allotted
 * block of memory (pointed to by 'pos') and return failed condition
 * to inform the calling module of memory shortage.
 * input:  mnemonic op_code, hex code, and output file pointer
 * output: execution outcome.
 */
int
fill_struct(char *name, char *code, FILE *fp)
{
	template	*pos;

	if ((pos = (template *)malloc(sizeof(template))) == (void *)NULL)
		return Failed;
	strcpy(pos->opcode, name);
	strcpy(pos->bin_equiv, code);

	/*
	 * Shouldn't be printing (calling out_put) here, but rather
	 * after the successful execution of 'build_tree'.  It is done
	 * so because the memory block pointed to by 'pos' will be
	 * freed inside 'build_tree'.
	 */
	out_put(pos, fp);

	if (build_tree(pos) == Failed)
	{
		free(pos);
		return Failed;
	}
	return Succeeded;
}

/*
 * Build an instruction tree.  If other data structures are used,
 * such as array, hashed table, etc., reprogram this function to
 * suit the target data structure.
 * input:  instruction node
 * output: outcome of building instruction tree call
 */
int
build_tree(template *tmp)
{
	/*
	 * set up head pointer in the first visit.  Afterwards,
	 * follow the head pointer to traverse the list.
	 * For now, the memory occupied by each instruction entry
	 * is freed to keep memory usage low.
	 */
	free(tmp);
	return Succeeded;
}
```

```
void
special_case(char *macro_name, FILE *fp)
{
	fprintf(fp, "loc: %*s\n", Size1, macro_name);
}

void
free_all(void)
{
	/* free all memory */
}

void
out_put(template *tmp, FILE *fp)
{
	fprintf(fp, "loc: %*s\t", Size1, tmp->opcode);
	fprintf(fp, "%*s\n", Size2, tmp->bin_equiv);
}
```