# Part 28: Adding More Run-time Flags

This part of our compiler writing journey really doesn't have anything to do with scanning, parsing, semantic analysis or code generation. In this part, I add the `-c`, `-S` and `-o` run-time flags to the compiler so that it behaves more like a traditional Unix C compiler.

So, if that's not interesting, feel free to skip to the next part of the journey.

## Compilation Steps

Up to now, our compiler has only been outputting assembly files. But there are more steps to convert a source code file in a high-level language to an executable file:

- Scan and parse the source code file to generate assembly output
- Assemble the assembly output to an object file
- Link one or more object files to produce the executable file

We've been doing the last two steps manually or with our Makefile, but I'm going to modify the compiler to call an external assembler and linker to perform the last two steps.

To do this, I'm going to rearrange some of the code in `main.c` and also write more functions in `main.c` to do the assembling and linking. Most of this code is typical string and file handling code done in C, so I'll go through the code but it may only be interesting if you've never seen this sort of code.

## Parsing the Command-Line Flags

I've renamed the compiler to be `cwj` to reflect the name of the project. When you run it with no command-line arguments, it now gives this usage message:

```
$ ./cwj
Usage: ./cwj [-vcST] [-o outfile] file [file ...]
       -v give verbose output of the compilation stages
       -c generate object files but don't link them
       -S generate assembly files but don't link them
       -T dump the AST trees for each input file
       -o outfile, produce the outfile executable file
```

We now allow multiple source code files as inputs. We have four boolean flags, `-v`, `-c`, `-S` and `-T`, and we can now name the output executable file.

The `argv[]` parsing code in `main()` is now changed to deal with this, and there are several more option variables to hold the results.

```
// Initialise our variables
O_dumpAST = 0;          // If true, dump the AST trees
O_keepasm = 0;          // If true, keep any assembly files
O_assemble = 0;         // If true, assemble the assembly files
O_dolink = 1;           // If true, link the object files
O_verbose = 0;          // If true, print info on compilation stages

// Scan for command-line options
for (i = 1; i < argc; i++) {
  // No leading '-', stop scanning for options
  if (*argv[i] != '-')
    break;

  // For each option in this argument
  for (int j = 1; (*argv[i] == '-') && argv[i][j]; j++) {
    switch (argv[i][j]) {
    case 'o':
      outfilename = argv[++i]; break;         // Save & skip to next argumen
    case 'T':
      O_dumpAST = 1; break;
    case 'c':
      O_assemble = 1; O_keepasm = 0; O_dolink = 0; break;
    case 'S':
      O_keepasm = 1; O_assemble = 0; O_dolink = 0; break;
    case 'v':
      O_verbose = 1; break;
    default:
```

```
        usage(argv[0]);
      }
    }
  }
```

Note that some options are mutually exclusive, e.g. if we only want assembly output with `-S`, then we don't want to link or create object files.

## Performing the Compilation Stages

With the command-line flags parsed, we can now run the compilation stages. We can compile and assemble each input file easily, but there may be a number of object files that we need to link together at the end. So we have some local variables in `main()` to store the object file names:

```
#define MAXOBJ 100
  char *objlist[MAXOBJ];        // List of object file names
  int objcnt = 0;               // Position to insert next name
```

We first process all the input source files in turn:

```
  // Work on each input file in turn
  while (i < argc) {
    asmfile = do_compile(argv[i]);      // Compile the source file

    if (O_dolink || O_assemble) {
      objfile = do_assemble(asmfile);   // Assemble it to object format
      if (objcnt == (MAXOBJ - 2)) {
        fprintf(stderr, "Too many object files for the compiler to handle\n");
        exit(1);
      }
      objlist[objcnt++] = objfile;      // Add the object file's name
      objlist[objcnt] = NULL;           // to the list of object files
    }

    if (!O_keepasm)                     // Remove the assembly file if
      unlink(asmfile);                  // we don't need to keep it
    i++;
  }
```

`do_compile()` has the code that used to be in `main()` to open the file, parse it ourselves and generate the assembly file. But we can't open up the hard-coded filename `out.s` like we used to; we now need to convert `filename.c` to `filename.s`.

## Altering the Input Filename

We have a helper function to alter filenames.

```
// Given a string with a '.' and at least a 1-character suffix
// after the '.', change the suffix to be the given character.
// Return the new string or NULL if the original string could
// not be modified
char *alter_suffix(char *str, char suffix) {
  char *posn;
  char *newstr;

  // Clone the string
  if ((newstr = strdup(str)) == NULL) return (NULL);

  // Find the '.'
  if ((posn = strrchr(newstr, '.')) == NULL) return (NULL);

  // Ensure there is a suffix
  posn++;
  if (*posn == '\0') return (NULL);

  // Change the suffix and NUL-terminate the string
  *posn++ = suffix; *posn = '\0';
  return (newstr);
}
```

Only the `strdup()`, `strrchr()` and the last two lines do any real work; the rest is error checking.

## Doing the Compilation

Here is the code that we used to have, now repackaged into a new function.

```
// Given an input filename, compile that file
// down to assembly code. Return the new file's name
static char *do_compile(char *filename) {
  Outfilename = alter_suffix(filename, 's');
  if (Outfilename == NULL) {
    fprintf(stderr, "Error: %s has no suffix, try .c on the end\n", filename);
```

```
    exit(1);
  }
  // Open up the input file
  if ((Infile = fopen(filename, "r")) == NULL) {
    fprintf(stderr, "Unable to open %s: %s\n", filename, strerror(errno));
    exit(1);
  }
  // Create the output file
  if ((Outfile = fopen(Outfilename, "w")) == NULL) {
    fprintf(stderr, "Unable to create %s: %s\n", Outfilename,
            strerror(errno));
    exit(1);
  }

  Line = 1;                     // Reset the scanner
  Putback = '\n';
  clear_symtable();             // Clear the symbol table
  if (O_verbose)
    printf("compiling %s\n", filename);
  scan(&Token);                 // Get the first token from the input
  genpreamble();                // Output the preamble
  global_declarations();        // Parse the global declarations
  genpostamble();               // Output the postamble
  fclose(Outfile);              // Close the output file
  return (Outfilename);
}
```

There's very little new code here, just the call to `alter_suffix()` to get the correct output file's name.

There is one important change: the assembly output file is now a global variable called `Outfilename`. This allows the `fatal()` function and friends in `misc.c` to remove assembly files if we never fully generated them, e.g.

```
// Print out fatal messages
void fatal(char *s) {
  fprintf(stderr, "%s on line %d\n", s, Line);
  fclose(Outfile);
  unlink(Outfilename);
  exit(1);
}
```

## Assembling the Above Output

Now that we have assembly output files, we can now call an external assembler to do this. This is defined as ASCMD in `defs.h`. Here's the function to do this:

```
#define ASCMD "as -o "
// Given an input filename, assemble that file
// down to object code. Return the object filename
char *do_assemble(char *filename) {
  char cmd[TEXTLEN];
  int err;

  char *outfilename = alter_suffix(filename, 'o');
  if (outfilename == NULL) {
    fprintf(stderr, "Error: %s has no suffix, try .s on the end\n", filename);
    exit(1);
  }
  // Build the assembly command and run it
  snprintf(cmd, TEXTLEN, "%s %s %s", ASCMD, outfilename, filename);
  if (O_verbose) printf("%s\n", cmd);
  err = system(cmd);
  if (err != 0) { fprintf(stderr, "Assembly of %s failed\n", filename); exit(1
  return (outfilename);
}
```

I'm using `snprintf()` to build the assembly command which we will run. If the user used the `-v` command-line flag, this command will be shown to them. Then we use `system()` to execute this Linux command. Example:

```
$ ./cwj -v -c tests/input54.c
compiling tests/input54.c
as -o  tests/input54.o tests/input54.s
```

## Linking the Object Files

Down in `main()` we build up a list of object files that `do_assemble()` returns to us:

```
        objlist[objcnt++] = objfile;      // Add the object file's name
        objlist[objcnt] = NULL;           // to the list of object files
```

So, when we need to link them all together, we need to pass this list to the `do_link()` function. The code is similar to `do_assemble()` in that it uses `snprintf()` and `system()`. The difference is that we must track where we are up to in our command buffer, and how much room is left to do more `snprintf()` ing.

```
#define LDCMD "cc -o "
// Given a list of object files and an output filename,
// link all of the object filenames together.
void do_link(char *outfilename, char *objlist[]) {
  int cnt, size = TEXTLEN;
  char cmd[TEXTLEN], *cptr;
  int err;

  // Start with the linker command and the output file
  cptr = cmd;
  cnt = snprintf(cptr, size, "%s %s ", LDCMD, outfilename);
  cptr += cnt; size -= cnt;

  // Now append each object file
  while (*objlist != NULL) {
    cnt = snprintf(cptr, size, "%s ", *objlist);
    cptr += cnt; size -= cnt; objlist++;
  }

  if (O_verbose) printf("%s\n", cmd);
  err = system(cmd);
  if (err != 0) { fprintf(stderr, "Linking failed\n"); exit(1); }
}
```

One annoyance is that I'm still calling the external C compiler `cc` to do the linking. We really should be able to break this dependency on another compiler.

A long time ago, it was possible to link a set of object files manually by doing, e.g.

```
$ ln -o out /lib/crt0.o file1.o file.o /usr/lib/libc.a
```

I assume that it should be possible to do a similar command on current Linux, but so far my Google-fu isn't enough to work this out. If you read this and know the answer, let me know!

## Losing `printint()` and `printchar()`

Now that we can call `printf()` directly in the programs that we can compile, we no longer need our hand-written `printint()` and `printchar()` functions. I've removed `lib/printint.c`, and I've updated all of the tests in the `tests/` directory to use `printf()`.

I've also updated the `tests/mktests` and `tests/runtests` scripts so that they use the new compiler command-line arguments, and ditto the top-level `Makefile`. So a `make test` still runs our regression tests OK.

## Conclusion and What's Next

That's about it for this part of our journey. Our compiler now feels like the traditional Unix compilers that I'm used to.

I did promise to add in support for an external pre-processor in this step, but I decided against it. The main reason is that I would need to parse the filenames and line numbers that the pre-processor embeds in its output, e.g.

```
# 1 "tests/input54.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "tests/input54.c"
int printf(char *fmt);

int main()
{
  int i;
  for (i=0; i < 20; i++) {
    printf("Hello world, %d\n", i);
  }
  return(0);
}
```

In the next part of our compiler writing journey, we will look at adding support for structs to our compiler. I think we might have to do another design step first before we get to implementing the changes. Next step