Abstraction and Filtration of GNU Sort

by Jin S. Choi and Philip Greenspun

This is an example of Abstraction and Filtration (in preparation for Comparison) of a program in the C language: GNU sort.

The function of the `sort` program is to take lines of data from one or more files, specified on the Unix command line, and output the sorted concatenation of the lines. The program also accepts options, e.g., which character to use as the delimeter between lines.

Algorithmically, the program makes use of the External R-Way Merge, which divides up the input into chunks that will fit into memory, sorts each chunk in memory using a merge sort and writes it to a temporary file, then merges the resulting chunks together. Merge sort is advantageous in this program over, say, Quicksort, because it uses fewer comparisons, it is stable, and it is more efficient when the input is already sorted.

Examples of using `sort`:

- `sort file1 file2 > file3`
  Lexicographically sort the concatenation of the lines in file1 and file2 and write it into file3. Lexicographic sort is a sort in "alphabetic order", with shorter strings coming before longer strings. "10" sorts before "2", for example.
- `sort -nr inputfile`
  Sort the input file in reverse numeric order, treating any leading digits as numbers, not lexicographic strings. Output the result to the console. "10" will be considered to come after "2", although in this case we are asking for *reverse* order, so it will appear first.

Abstraction Level 0

The 4626 lines of source code in the C programming language.

Abstraction Level 1

We take the raw source code, including comments, and break it down into functional pieces. For this example, we look at the `sequential_sort` function which does the actual work of sorting lines. In this case, the comments specify all the requirements for the inputs, give a reference to the algorithm being employed, with a helpful notation indicating that a certain optimization was employed. The basic algorithm uses the technique of mathematical induction.

```
/* Sort the array LINES with NLINES members, using TEMP for temporary space.
   Do this all within one thread.  NLINES must be at least 2.
   If TO_TEMP, put the sorted output into TEMP, and TEMP is as large as LINES.
   Otherwise the sort is in-place and TEMP is half-sized.
   The input and output arrays are in reverse order, and LINES and
   TEMP point just past the end of their respective arrays.

   Use a recursive divide-and-conquer algorithm, in the style
   suggested by Knuth volume 3 (2nd edition), exercise 5.2.4-23.  Use
   the optimization suggested by exercise 5.2.4-10; this requires room
   for only 1.5*N lines, rather than the usual 2*N lines.  Knuth
   writes that this memory optimization was originally published by
   D. A. Bell, Comp J. 1 (1958), 75.  */

static void
sequential_sort (struct line *restrict lines, size_t nlines,
                 struct line *restrict temp, bool to_temp)
{
  if (nlines == 2)
    {
```

Base case.

```
      /* Declare `swap' as int, not bool, to work around a bug
         <http://lists.gnu.org/archive/html/bug-coreutils/2005-10/msg00086.html>
         in the IBM xlc 6.0.0.0 compiler in 64-bit mode.  */
```

`swap` is 1 if the first line is greater than the second line, 0 otherwise. Note from the first comment that the `lines` array is structured backwards from what we would expect, so negative indexes are used.

```
      int swap = (0 < compare (&lines[-1], &lines[-2]));
```

Swap the lines if called for, into the temp space if requested, or in place otherwise.

```
      if (to_temp)
        {
```

swap is 0 or 1, so -1 - 0 = -1, -2 + 0 = -2, and -1 - 1 = -2, -1 + 1 = -1

```
          temp[-1] = lines[-1 - swap];
          temp[-2] = lines[-2 + swap];
        }
      else if (swap)
        {
          temp[-1] = lines[-1];
          lines[-1] = lines[-2];
          lines[-2] = temp[-1];
        }
    }
  else
    {
```

Inductive step.

Split the lines in half.

```
      size_t nlo = nlines / 2;
      size_t nhi = nlines - nlo;
      struct line *lo = lines;
      struct line *hi = lines - nlo;
```

Sort the high half, which is guaranteed to contain at least two lines, as required (if there were two lines, we wouldn't be here. If there are 3 lines, because of the way we split in half, two lines will go in the high array and one in the low. If there are more than 3 lines, then obviously nhi will be more than 2).

```
      sequential_sort (hi, nhi, temp - (to_temp ? nlo : 0), to_temp);
```

Sort the low half if necessary.

```
      if (1 < nlo)
        sequential_sort (lo, nlo, temp, !to_temp);
      else if (!to_temp)
        temp[-1] = lo[-1];
```

Merge the high and low buffers to the destination.

```
      struct line *dest;
      struct line const *sorted_lo;
      if (to_temp)
        {
          dest = temp;
          sorted_lo = lines;
        }
      else
        {
          dest = lines;
          sorted_lo = temp;
        }
      mergelines (dest, nlines, sorted_lo);
    }
}
```

Abstraction Level 2

For a higher level abstraction, we divide the single file of source code into two sections: (1) headers and definitions; (2) functions. Within each section, we summarize the purpose of each block of code.

Headers and Definitions

- copyright
- inclusion of standard system headers and references to files that define data structures and functions outside of this file
- definitions of constants and macros to account for variations in the target systems
- definitions of constants to assign symbolic names to various parameters used by the program
- definition of data structures that will be used by the algorithm
  - lines
  - buffers
  - key fields
  - months
  - merge tree nodes
  - merge node priority queues
  - various lookup tables
    - character-type tables

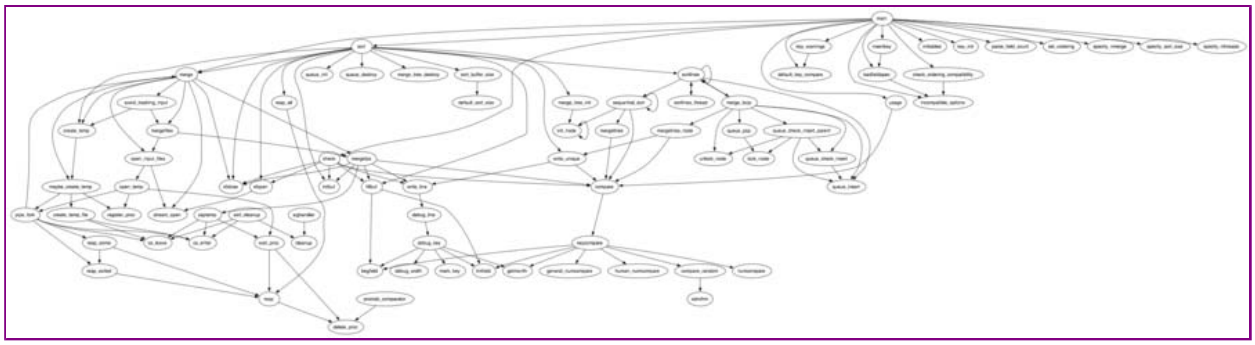
        - months
        - command-line option table
    - temporary files
    - input files
    - hash tables

Functions

Most programs are structured using functions to package code into manageable units. In many cases, they can be treated as the functional modules we are trying to identify, although since functions can call other functions, not all functions are at the same level of abstraction. There will be many small functions that abstract small units of code, and some larger ones that call upon the smaller ones to carry out operations at a higher level of abstraction. The size of a function can serve as a general indication of the level of abstraction, although it is not a perfect metric. Leaving the question of where to draw the line at various levels of abstraction for later, here are some of the procedures used, grouped by type.

- utility procedures
    - die - exists the program
    - usage - outputs a message to the user detailing the existing options
    - cs_enter, cs_leave - procedures to lock critical sections of code that must not be interrupted
    - proctab_hasher, proctab_comparator - used for hash table implementation
    - reap, register_proc, delete_proc, wait_proc,
    - reap_existed, reap_some, reap_all, cleanup, exit_cleanup - process accounting
    - create_temp_file, maybe_create_temp, create_temp, open_temp, add_temp_dir, zaptemp - temp file creation and management
    - stream_open, xfopen, xfclose - opening and closing files
    - dup2_or_die, pipe_fork - process management
    - inittables - initialization of necessary lookup tables
    - specify_nmerge, specify_sort_size, specify_nthreads, default_sort_size, sort_buffer_size - calculation of various parameters, determined in part by user inputs
    - buffer_linelim - utility procedure for working with buffers
    - find_unit_order, human_numcompare, numcompare, general_numcompare, getmonth - parse numeric options and months, compare numbers
    - random_md5_state_init - initialize random state for MD5
    - xstrxfrm - locale transformation
    - compare_random - compare strings using a hash function
    - debug_width, mark_key, debug_key, debug_line - debugging
    - write_line - output a line
    - avoid_trashing_input - avoid inadvertently overwriting an input file
- core procedures
    - initbuf - initialize a buffer
    - begfield, limfield - calculates the location and limit of a specified fieldwithin a line
    - fillbuf - fill a buffer from a file
    - compare - compare two lines
    - check - check for pre-ordered files
    - open_input_files - opens input files
    - mergefps - merge and output lines from given file pointers
    - mergefiles - calls mergefps to merge and output lines from given files
    - mergelines - merge arrays of lines into one array
    - sequential_sort - uses a sequential sort to sort an array of lines, calling itself recursively, and mergelines
    - merge_tree_init, merge_tree_destroy, init_node, compare_nodes, lock_node, unlock_node - management of merge tree data structures
    - queue_destroy, queue_init, queue_insert, queue_pop, queue_check_insert, queue_check_insert_parent - manages merge node queue structures
    - write_unique - output a line to a temporary file unless user has requested unique output and line is not unique
    - mergelines_node - merge the lines for a node in a merge tree
    - merge_loop - works through a merge queue, merging nodes until the end is reached
    - sortlines_thread, sortlines - sorts lines, possibly in parallel, using sortlines itself, sequential_sort, and merge_loop
    - merge - merge input files into output file
    - sort - sort input files into output file
    - key_numeric, default_key_compare, key_to_opts, key_warnings, keycompare, key_init,
    - insertkey, badfieldspec, incompatible_options, check_ordering_compatibility, parse_field_count, set_ordering - key and argument handling
    - main - options handling, setup of input and output files, do merge or sort by calling merge() or sort()

It is difficult from a linear list of procedures to get an idea of the flow of the program. To better percieve the structure, we can generate a call-graph, detailing which functions are called by which others. If we layout the graph hierarchically so that the callers are generally above the functions that they call, the ones towards the top will tend to be the abstract modules while the ones nearer the bottom will be smaller ones providing detailed bits of functionality. In many well-structured programs, the code itself will reveal the levels of abstraction we are looking for.



This call graph contains only function calls which reference functions within the file itself. All references to system or library calls have been filtered out.

Abstraction Level 3: modules

- options parsing
- file handling
- threading
- reading files into buffers and converting raw data into lines
- binary merge trees and priority queues for n-way file merge
- merge sort of individual lines

Abstraction Level 4: behavior

The final level of abstraction is the overall behavior of the program. The best way to understand how the program is used is to refer to its documentation. In this case, the command-line options constitute the entirety of this program's user interface, and their documentation provides the key to understanding this program as an expressive work.

Filtration of Abstraction Level 1

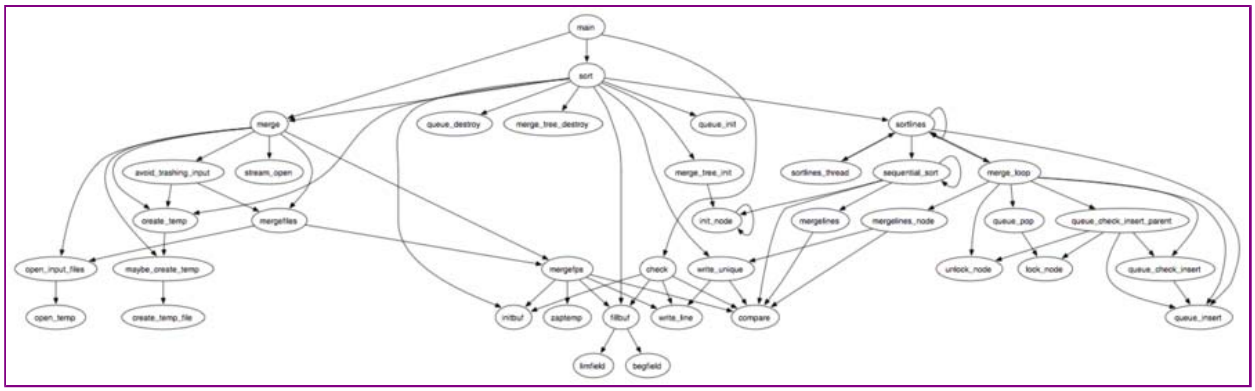
The entirety of the code given in the sample is central to the program. However, it is based on a published algorithm, using a modification for efficiency that also comes from the literature. All of it could be filtered out to its being an expression of an idea in the public domain, yet there are features that argue for retention. For example, the simple matter of the selection of merge sort in this context over other available algorithms; the use of a memory optimization not often seen; most strikingly, the use of a reverse-ordered array to store the lines. The last is a result of the method used to read lines into memory without overrunning the space available.

This function does not include any library or system calls (functions provided by the system) that should be filtered.

Filtration of Abstraction Level 2

At this stage, we filter out all the implementation details that deal with options parsing, file handling, and threading as they are elements dictated by external factors, namely, the usage of this program within the context and tradition of the Unix command-line environment. These modules deal with functionality necessary to many command-line programs and are not unique to this program, although details of the implementation such as error handling, the particular command-line options used and how they affect the behavior of the program, and how the threads are used might be considered distinctive elements, and subject to comparison. Similarly, binary trees and priority queues and the algorithms used to manipulate them are well-understood and common in the field, with many implementations described in the public domain, so the details of their implementation could be filtered. They way in which they are used in this program to distribute the work efficiently to multiple threads, however, is the core to this program, so it is at this level of abstraction that we can begin the comparison stage.

If we redo the call-graph by removing all the functions that can be deemed to be dictated by external factors, as listed above, we come up with something like this:



Already from this, we have a good idea for the structure of the program. A full understanding of the methods used requires more than a call-graph; we would have to run through an execution of the program and describe how each element is used. However, that may not be necessary for comparison purposes. We can briefly describe the overall operation of the program as "sort lines from input files by mergesort, using temporary files as necessary." Any necessary comparison step would proceed by doing a detailed analysis of the portions listed above.

Filtration of Abstraction Level 3

As discussed above, the implementation of options parsing, file handling, and threading are all dictated by environmental factors, and should be filtered. They are included in the list for completeness. This leaves the implementation of the modules core to the program:

- ~~options parsing~~
- ~~file handling~~
- ~~threading~~
- reading files into buffers and converting raw data into lines
- binary merge trees and priority queues for n-way file merge
- merge sort of individual lines

Filtration of Abstraction Level 4

The user interface of GNU sort relies heavily on the custom and tradition of UNIX command-line programs, providing for the use of shorthand options preceded with a single dash, or verbose options preceded with two dashes. The actual options and their behavior define the user interface of the program, and are what should be considered in the comparison phase.

As GNU sort is an independent implementation of a line of line-sorting programs, beginning with `sort` from AT&T System V, to BSD `sort`, we can begin to filter this abstraction level by removing what it shares in common with its predecessors. That is difficult to do without delving into the history of sort programs, but we can look at a modern version of one of its ancestors, the `sort` program from FreeBSD. Many of the options are identical, both due to common ancestry and perhaps subsequent cross-pollination of ideas between the lines. In any case, we can be sure that any unique options represent unique features of this implementation (`--compress-program`, for example).

---

jsc@alum.mit.edu