/*

Hi,

just to mention... it's Sept. 2011 now. Some changes in Makefile and manpage were added, as well as these some words now. And also a PKGBUILD for Arch Linux is available. But it will not be inside the tgz-file. I will upload it to AUR, and maybe also put on my homepage.

Have fun on Arch. :-)

(Mo 5. Sep 02:17:55 CEST 2011)

===

Hello, Dear Reader!

It's April, 1st, 2007
and here is an update of multiple! :)

No, this is not a joke.
I added a struct-member in the comparison-struct,
and this member contains the stat-mode of the file.

This reduces disk-accesses, because the selected files
in the version 0.2 were lstat()-ed again, because
the status was not saved in a data structure.

So now the program should be slightly faster.
But the effect seems to be a joke, because
it has not a big impact on the speed.
The reason is: there are under normal conditions not
so much files that are equal. So, the second lstat()-call
will only be done on a small subset of the files
that were stat()ed the first time.

But it feels better to add this small thing,
because it was planned since many years.
I didn't had the time and motivation
to do it for a long time.
So I did it today.

(BTW: I doubt there will be newer versions of multiple,
because I switched to OCaml; the newer versions then
will be called "omultiple", where the "o" stands for
OCaml. In the OCaml-versions I have switched to the
recommended idea of using hash-functions instead of
comparing byte-by-byte.
If you want a byte-by-byte comparison, you should
use this good old multiple tool. :)

Otherwise I can recommend the "omultiple" program.
It will be much faster. :))

(Mo 5. Sep 02:53:24 CEST 2011)

Overview about *multiple*

multiple: a tool for detecting files with equal contents

Hello!

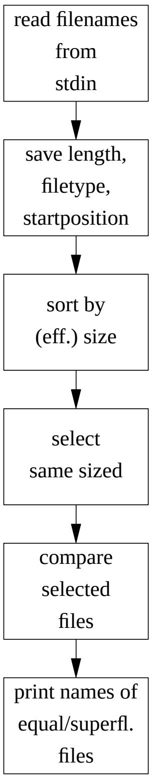
This program is used to find files, which contains the same contents/data, but maybe has different names. Different names hereby can be the same basename, but the file ist located in other directories (that means "filename" is the full path to the file), or even different basenames. No matter, what really the case is: the files will be viewed inside an then it's decided, if they were equal or not. The name is *not* the decision making property. It's the *contents.*

Why this programm?

I wrote it, because I often does exporting interesting articles from my newsreader. I stored them under different names, but if I stored whole usenet-threads more then one time, I did often store the same articles more then once.

These files had to be compared and all superflous files had to be recognized (print names or delete them). And so the idea of multiple grows.

Now let's have a look on the way, the data will be processed, from piping in the filenames to print out the names of the files, detected as being equal:


read filenames from stdin
save length, filetype, startposition
sort by (eff.) size
select same sized
compare selected files
print names of equal/superfl. files

The sort by size is done, because this preselection assumed to me an effective way of reducing the amount of data (filenames/files), which had to be compared.

On the first version of multiple I got some critique via usenet. Some things doesn’t match because of an other focus of the tool (it‘s a tool for ad-hoc-using, not to be constraint to use a md5sum-database of all files, which should be compared, for example).

But there was more efficient critique, on the first version of multiple, which has encouraged me to write the version 0.2 of multiple. It was a matter of honour, to work the new things in:
I worked on the sorting algorithm (using qsort now, and selecting the same sized-files out of the list of sorted-by-size files) and this has increased the performance well.

But, well... lets have a look into the sources now.

```
*/


/*
#define TEST
*/

/* ================================================================ */
/* multiple.c:  Select Filenames of Files that contains equal data. */
/* ---------------------------------------------------------------- */
/* ---------------------------------------------------------------- */
/* Written and Copyright: Oliver Bandel, oliver@first.in-berlin.de  */
/* ---------------------------------------------------------------- */
/* This Software underlies the GNU-GPL                              */
/* ---------------------------------------------------------------- */
/* Have a lot of fun with it :-)                                    */
/* ================================================================ */

#define VERSION "Version 0.5.2 ( Mo 5. Sep 02:53:24 CEST 2011 )"



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

#define MAX_LINE_LENGTH 1024
#define INPUT_BUFFER (MAX_LINE_LENGTH * sizeof(char))


enum { NO = 0, YES };


/* ENUM’s für file_registry() */

enum what_to_do { REGISTER,
                  DELETE,
                  GET_SAME_SIZED_LIST,
                  CHECK_SAME_FILENAMES
                };


/* ENUMS für state’s von  compare_same_sized() */

enum{
      START,
      A,
      B,
      C,
      D,
      F
    };



/* Type-Definitions, Structs and Unions */

typedef off_t COUNTER;

typedef struct {
                 char *          name;
                 off_t           size;           /* filesize */
                 off_t           headerlength;   /* für Option "-i" */
                 off_t           eff_size;       /* size - headerlength */
                 mode_t          stat_mode;
               } FILEINFO;


typedef struct {
         FILEINFO*  base;
         COUNTER    nmemb;
       } FILELIST;
```

```
typedef struct {
                 COUNTER     start;
                 COUNTER     stop;
               } INDIZES;



typedef struct {
                 short   ignore_headers;            /* -i */
                 short   print_all_files;           /* -a */
                 short   print_version;             /* -v */
                 short   print_help_message;        /* -h */
                 short   zero_bytes_files_ok;       /* -0 */
                 short   delete_superflous;         /* -d */
               } CLI_OPTIONS;

typedef struct {
                  off_t   length;
                  int     status;
               } GET_LENGTH;


/* List for file-comparison (for comparing files; correct english?) */

typedef struct _file_cmp FILE_CMP;

struct _file_cmp {
                   char*       fname;       /* filename */
                   FILE_CMP*   next_cmp;    /* next entry to compare */
                   FILE_CMP*   next_clone;  /* next entry to clone   */
                   off_t       cmp_position;/* if Option "-i": start cmp here */
                   mode_t      stat_mode;
                 };

/* List for equal files (for equal size or really equal files; you need to know the context) */

typedef struct _eq_list EQ_LIST;

struct _eq_list {
                  char*      fname;         /* filename */
                  EQ_LIST*   file_list;     /* this list with equal files */
                  EQ_LIST*   next_eq_list;  /* next list with equal files */
                };



/* Definition of global variables */
/* ------------------------------ */

CLI_OPTIONS cli_options = {0, 0, 0, 0, 0, 0}; /* Default’s setzen */
char* myopts = "aiv0hd"; /* accepted options */


/* Function-Declarations */
/* --------------------- */


void     register_stdin_files();

FILE_CMP** file_registry( FILEINFO* infoptr, int todo );
FILE_CMP*  create_list( FILEINFO* this, COUNTER start, COUNTER end );

void     print_usage( FILE* outputstream );
int      print_help_message( FILE* outstream );
int      get_cli_options( CLI_OPTIONS* options, int argc, char* argv[] );
int      compare_same_sized( FILE_CMP** liste ); /* check list for comparision*/
int      are_filesequal(FILE_CMP* a, FILE_CMP* b); /* compare files bytewise */

void     print_all_multiple_files( FILE_CMP** liste );
void     print_superflous_files( FILE_CMP** liste );

GET_LENGTH  read_until_body( char* filename ); /* result: headerlength of file */

__inline__
int cmp_filesize( const void* a, const void* b );





/*
```

Hi! This is the main Program
Function *main*

Here we are looking for the arguments from the command-line-interface. If the data (filenames) come in via Pipe, all is ok and then we can start working. If the data will not be piped in, than an error message occurs and the user get a short description, how to get a detailed help.

```
*/
/* ================================================================ */
/* ================================================================ */
/* ================================================================ */
int main( int argc, char* argv[] )
{
  extern CLI_OPTIONS cli_options; /* the global variable for the CLI-options */
  FILE_CMP**         samesized;


  /* get command line information and set members of "cli_options"-structure */
  /* ---------------------------------------------------------------------- */
  get_cli_options(&cli_options, argc, argv);

  if( cli_options.print_help_message ) /* help-message wanted ? */
  {
    print_help_message(stdout);
    exit(EXIT_SUCCESS);
  }

  if( cli_options.print_version )      /* version-information wanted ? */
  {
    printf("%s\n", VERSION);
    exit(EXIT_SUCCESS);
  }


  /* if tty => exit; if information readable via pipe, read it */
  /* -------------------------------------------------------- */
  if( isatty(STDIN_FILENO) )
  {
    fprintf(stderr, "‘multiple’ Error: Pipe filenames in, e.g. \"find <dirname> | multiple <options>\"\n");
    print_usage(stderr);
    exit(EXIT_FAILURE);
  }
  else
  {
    register_stdin_files(); /* read filenames from stdin and remember them */
  }


  /*
     *=> now the files will be sorted by size for preselection
     *=> thereafter they the files with same size will be selected
  */

  samesized = file_registry(NULL, GET_SAME_SIZED_LIST);   /* create list of samesized files */

  compare_same_sized( samesized );   /* Pointer entsprechend umbauen */


/* hier duerfte die Variable nicht mehr "samesized" heissen, sondern */
/* z.B. sorted oder so. */
/* WAS IST MIT DER DOPPELTEN-Files-Liste?  => free() ?!    */
/*                                         => samesized    */
/*  => file_registry( NULL, DELETE ); .... reicht das aus? */


  /* After all that sorting stuff, now only the right things must be printed */
  /* ---------------------------------------------------------------------- */
  if( cli_options.print_all_files )
  {
    print_all_multiple_files( samesized );
  }
  else
  {
    print_superflous_files( samesized );
  }

  file_registry( NULL, DELETE ); /* cleaning up */

  return(0);
}



/*
```

Function *register_stdin_files*

In this function, we read in all filenames, that came in via pipe. And we save the names for future use. Before we registering the files, we are looking for the filetype.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
void register_stdin_files()
{
  extern CLI_OPTIONS cli_options;

  FILEINFO     info;
  struct stat  buf;

  int     ret;
  char*   line;


  line = malloc(INPUT_BUFFER);
  if( line == NULL )
  {
    fprintf(stderr, "Can’t allocate memory.\n");
    exit(EXIT_FAILURE);
  }
    /* only: - regular Files
             - Links to existing regular files
    */

    /* reading the lines from stdin */
    /* ============================ */
    while((fgets(line, INPUT_BUFFER, stdin)) != NULL)
    {
      line[strlen(line)-1] = ’\0’;  /* remove "\n" at end of line (chop()) */

      /* get the file-infos from stat-structure */
      /* -------------------------------------- */
      ret = stat(line, &buf);
      if( ret )
      {
        if( lstat(line, &buf) ) /* we really have a problem here ! */
        {
          fprintf(stderr, "lstat-error on file %s\n", line);
          continue;
          exit(EXIT_FAILURE);
        }

        /* ignore broken symbolic links, inform the user */
        /* --------------------------------------------- */
        fprintf(stderr, "broken symbolic link: %s\n", line); /* message   */
        continue;                                             /* next file */
      }


      /* if the file is not a regular file, ignor it (next file, please!) */
      /* ---------------------------------------------------------------- */
      if( ! S_ISREG(buf.st_mode) )
      {
        continue; /* next file */
      }


      /* next file if this one is 0 bytes long and not option "-0" is used */
      /* ----------------------------------------------------------------- */
      if( ! cli_options.zero_bytes_files_ok && buf.st_size == 0 )
      {
        continue;
      }


      info.name       = line;
      info.size       = buf.st_size;
      info.stat_mode  = buf.st_mode; /* remember the mode: not again accessing filesystem for that later */

      /* now we notice the length of file and startposition for comparison */
      /* ----------------------------------------------------------------- */
      if( cli_options.ignore_headers ) /* IGNORE HEADERS: compare bodies */
      {
        GET_LENGTH  res; /* block-lokal varable */

        res = read_until_body(info.name); /* get length/status value */

        if(res.status) /* no header found => binary file => ignore it! */
        {
          fprintf(stderr, "No Header in file %s, ignoring file.\n", info.name);
          continue; /* next file */
        }

        /* a length of header was detected */
        /* ------------------------------- */
        info.headerlength = res.length;                    /* compare-startpos*/
        info.eff_size     = info.size - info.headerlength; /* => body-size */

      }
      else /* DON’T IGNORE HEADERS for comparision: COMPARE FROM FIRST BYTE */
      {
        info.headerlength = 0;
        info.eff_size     = info.size;
      }

      /* OK, registering the file (remember it ) */
      /* --------------------------------------- */
      file_registry( &info, REGISTER );
    }

  return;
}
/*
```

Function *file_registry*

This function is the function, that registers the files. It also has the ability and to sort them by size.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
FILE_CMP** file_registry( FILEINFO* infoptr, int todo )
{
  static COUNTER        num;
  /* static COUNTER mem_num; */ /* ist vorgesehen für seltneres realloc() */

  COUNTER        i;

  static void*   ptr;
  FILEINFO*      this;
  /*
  FILELIST       retval;
  */


  this = NULL;
  /* Maybe that makes sense for future versions....
  retval.base  = NULL;
  retval.nmemb = 0;
  */


  if( ptr == NULL && todo != REGISTER )
  {
    fprintf(stderr, "file_registry(): No Files registrered => can’t do Operation.\n");
    exit(EXIT_FAILURE);
  }


  /* Select the functionality */
  /* ------------------------ */
  switch (todo )
  {

/* reg_register_file() */
    case REGISTER:
    {
      ++num;

      /* Optimieren: Es wird jedesmal Speicher allokiert! => Chunks besser */

      ptr = (ptr == NULL) ? malloc(sizeof(FILEINFO)) : realloc( ptr, num * sizeof( FILEINFO ) );
      if( ptr == NULL )
      {
        fprintf(stderr, "no memory available.\n");
        exit(EXIT_FAILURE);
      }

      this = ptr + (num-1) * sizeof(FILEINFO);

      memcpy( this , infoptr, sizeof( FILEINFO ) );
      this->name = strdup( infoptr->name );

      if( this->name == NULL ) /* hat wohl kein mem bekommen */
        exit(EXIT_FAILURE);
    }
    break;



/* reg_delete_filelist() */
    case DELETE:
    {
      for( i = 0; i < num; i++ )
      {
        this = ptr + i * sizeof(FILEINFO);
        free( this->name );
      }
      free(ptr);
      ptr = NULL;
      num = 0;
    }
    break;


/* reg_check_filenames() */
    case CHECK_SAME_FILENAMES: /* Falls filenamen mehrfach vorhanden sind, hier löschen */
    {
      /* wenn alle filenames nur einmal eingepipet werden, ist dies hier */
      /* nicht notwendig; kann man ja als extra-option (Option "-c" für "check filenames") */
      /* noch nachimplementieren. */

      /*
      qsort(ptr, num, sizeof(FILEINFO), cmp_filename );
      doppelte_eliminieren();
      */

    }
    break;
/* ############################################################################ */
/* reg_select_samesized() */
    /* gleich große files raussuchen; files mit uniq Größe verwerfen */
    /* ------------------------------------------------------------------- */
    case GET_SAME_SIZED_LIST:
    {
      off_t          start_size;
      /*
      char*          start_name;
      */
      COUNTER        start_index;
      COUNTER        last_index;
      COUNTER        counter;
      COUNTER        anzahl;
      FILE_CMP**     lptr;

      /* sort by filesize */
      /* ---------------- */
      qsort(ptr, num, sizeof(FILEINFO), cmp_filesize );


      /* create list          */
      /* ==================== */
      lptr = calloc( (num/2 + 2), sizeof(FILE_CMP*));  /* Mem für num Zeiger allok. */
      if( lptr == NULL )
      {
        fprintf(stderr, "no memory available!\n");
        exit(EXIT_FAILURE);
      }
      anzahl = 0;

      start_size = ((FILEINFO*)ptr)->eff_size;
      //start_name = ((FILEINFO*)ptr)->name; // use later

      counter     = 1;
      start_index = 0;
      last_index  = 0;


      /* looking for files with sme size in the list of by-size-sorted */
      /* ------------------------------------------------------------- */
      for( i = 0; i < num; i++, counter++, this++ )
      {
        this = (FILEINFO*)ptr + i;

        if( start_size != this->eff_size )
        {
          last_index  = i - 1;

          if( counter > 1  && start_index != last_index ) /* GLEICH GROSSE! */
          {
            *(lptr + anzahl) = create_list( ptr, start_index, last_index);
            ++anzahl;
          }

          start_size = this->eff_size;

          start_index = i;
          counter = 0;
        }
        else
        {
          last_index = i;
        }
      }


      if( counter > 1 ) /* NOCH MEHR GLEICH GROSSE files */
      {
        *(lptr + anzahl) = create_list( ptr, start_index, last_index);
        ++anzahl;
      }
      /* else { printf("NUR NOCH EINE EINZELNE!\n"); } */


      /* shortening  */
      /* ----------- */
      lptr = realloc(lptr, (anzahl + 1) * sizeof(FILE_CMP*));
      if( lptr == NULL )
      {
        fprintf(stderr, "no memory available.\n");
        exit(EXIT_FAILURE);
      }
      lptr[anzahl] = NULL; /* END-MARKE */

      return lptr;
    }
    break;

/* ############################################################################ */


  }
  return NULL;
}



/*
```

Function *cmp_filesize*

This function is for sorting the files by size. It’s used/invoked by the "sort"-function. Because this function will be invoked for each comparison, it’s declared as an inline function (to increase performance).

```
*/
/* ================================================================= */
/* ==== sort by size =============================================== */
/* ================================================================= */
__inline__
int cmp_filesize( const void* a, const void* b )
{
  FILEINFO* aa = (FILEINFO*) a;
  FILEINFO* bb = (FILEINFO*) b;

  if( aa->eff_size == bb->eff_size )
    return 0;

  return ( aa->eff_size  >  bb->eff_size  ?  1  :  -1 );
  return 0;
}


/*
```

Function *get_cli_options*

This function parses the parameters from the command-line-interface and sets the option-values.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
int get_cli_options( CLI_OPTIONS* options, int argc, char* argv[] )
{
  int c;
  extern char* myopts;
  extern void  print_usage();

  while((c = getopt( argc, argv, myopts ) ) != -1 )
  {
    switch( c )
    {
      case ’?’: print_usage(stderr); exit(EXIT_FAILURE);
      case ’i’: options->ignore_headers  = 1;        break;
      case ’a’: options->print_all_files = 1;        break;
      case ’v’: options->print_version   = 1;        break;
      case ’h’: options->print_help_message = 1;     break;
      case ’0’: options->zero_bytes_files_ok = 1;    break;
      case ’d’: options->delete_superflous = 1;      break;
    }
  }
  if( options->delete_superflous && options->print_all_files )
  {
    fprintf(stderr, "Error: contradiction: don’t use both Options -d and -a\n");
    exit(EXIT_FAILURE);
  }

  return 0;
}


/*
```

Function *create_list*

Here a list with files of same size will be created.

```
*/
/* ================================================================= */
/* Erzeugen der Liste mit den gleich grossen Files...=============== */
/* ----------------------------------------------------------------- */
/* ----------------------------------------------------------------- */
/* Bedenken: start und end sind Index-Werte! => "k <= ..."           */
/* ================================================================= */
FILE_CMP* create_list( FILEINFO* this, COUNTER start, COUNTER end )
{
  FILE_CMP* tmplist;
  FILE_CMP* eqlist;
  COUNTER   counter = end - start + 1;


  this += start; /* Auf den Anfang setzen */

  tmplist = calloc((end - start + 1), sizeof(FILE_CMP));
  if( tmplist == NULL )
  {
    fprintf(stderr, "no memory available\n");
    exit(EXIT_FAILURE);
  }
  eqlist = tmplist; /* merke Anfang! */

  while( counter-- ) /* for all files put entries into FILE_CMP-Structure */
  {
    tmplist->fname = this->name;
    tmplist->cmp_position = this->headerlength;
    tmplist->stat_mode = this->stat_mode; /* we need this later: decision if file is superflous */

    if( counter >= 1 )
      tmplist->next_cmp = tmplist + 1;

    tmplist++;
    this++;
  }
  return eqlist;
}



/*
```

Function *print_all_multiple_files*

After the work of detecting is done, we here going to print ALL the files, wich are detected as being multiple.

```
*/
/* ================================================================ */
/* ================================================================ */
/* ================================================================ */
void print_all_multiple_files( FILE_CMP** liste )
{
  FILE_CMP* start;
  FILE_CMP* traverse;
  off_t     i;

  i = 0;
  while((start = *(liste+i++)))
  {
    do
    {
      traverse = start;

      if( traverse->next_clone == NULL ) /* next horizontal (next_cmp), if no CLONE */
        continue;

      do
      {
        printf("%s ", traverse->fname );
      }while((traverse=traverse->next_clone));

      printf("\n");

    }while((start=start->next_cmp));
  }
  return;
}


/*
```

Function *print_superflous_files*

If we only want the names of superflous files, then we get them via this function.

We want to be sure, that if there are regular files and even symbolic links in the list of files, that we select a regular file as the one, we dont print out; so we are shure, that we dont delete the original file and the remaining files may only be broken symbolic links thereafter (if we delete the superflous).
If all files are symbolic links, then it does’t matter.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
void print_superflous_files( FILE_CMP** liste )
{
  FILE_CMP* start;
  FILE_CMP* traverse;
  FILE_CMP* regular_file;
  off_t     i;


  traverse = NULL;

  i = 0;
  while((start = *(liste+i++)))  /* for all samesized files */
  {
    do  /* for one of the sizes */
    {
      if( start->next_clone == NULL ) /* next horizontal (next_cmp), if no CLONE */
        continue;

      regular_file  = NULL;    /* no regular file now */
      traverse      = start;

      /* search regular file in list of clones */
      /* ------------------------------------- */
      do  /* vertical (cluster of files with same_contents) */
      {
        if( S_ISREG(traverse->stat_mode)) /* wenn es der regular File ist... */
        {
          regular_file = traverse;
          break;  /* nur das erste regular file! */
        }
      }while((traverse=traverse->next_clone)); /* move all clones_down */


      /* PRINT/DELETE REST of files */
      /* -------------------------- */
      traverse = start;

      if( regular_file == NULL ) /* if no regular file: print name of one link and throw it away  */
      {
        printf("%s", traverse->fname);   /* print name of NON-deleted files */
        traverse = traverse->next_clone; /* next file => ignore regular file */
      }
      else /* there is regular file in list: print it’s name; it will be ignored in delete-loop */
      {
        if( cli_options.delete_superflous )
          printf("%s", regular_file->fname); /* don’t delete regular file; print it’s name */
      }

      do   /* REST of: vertical (cluster of files with same_contents) */
      {
        /* if regular file, then ignore it here */
        /* ------------------------------------ */
        if( traverse == regular_file )
          continue;

        /* print or delete _rest_ of the files */
        /* ----------------------------------- */
        if( cli_options.delete_superflous )
          remove( traverse->fname );
        else
          printf("%s ", traverse->fname );

      }while((traverse=traverse->next_clone)); /* move all clones_down */

      printf("\n");

    }while((start=start->next_cmp)); /* horizontal move */
  }
  return;
}



/*
```

Function *compare_same_sized*

Here the main work will be done. It’s the main sorting-algorithm, which is implemented here.

There are some things, which we have pay attention to: first of all the files are sorted by filesize. That’s, because files can only be equal, if they are samesized. (well, if we ignore mailheaders, thats a special case...). That preselection is done, if we are at this point in the program, where we are now: Here, where we are comparing the preselected files.

When we have samesized files/preselected files, we cannot only compare one file to an other file, because of performance-reasons.

So, in the first version of multiple I created this scheme for choosing, which files I will compare with which others:

	file 1	**file 2**	**file 3**	**file 4**	**file 5**
file 1	---	---	---	---	---
file 2	compare	---	---	---	---
file 3	compare	compare	---	---	---
file 4	compare	compare	compare	---	---
file 5	compare	compare	compare	compare	---

With this scheme we don’t have to compare all files with all files. It works so: .....

But even if this seems to be ok, there are two problems:

- xxx
- Not all cases of equality are detected: if there are different classes of files, where the files of each class are different, but the files in each class are equal, then we can’t detect all classes!
  We stop after finding all equal files of the first class, detected as containing equal files...

The last problem I first find out, after working on the faster algorithm for version 0.2 of multiple.
I had thought again on the algorithm and the problem at all, and then discovered the "holes" in the comparison-algorithm.

(So, you see, that it’s sometimes better to forgot a problem and think of it to be solved and some time later take it up again and do some better solutions. A good beer needs seven minutes, one says in germany, and even a good program may take a while ;-))

Another thing is: When we found files, which are equal sized, we have to check, if they differ in the contents.

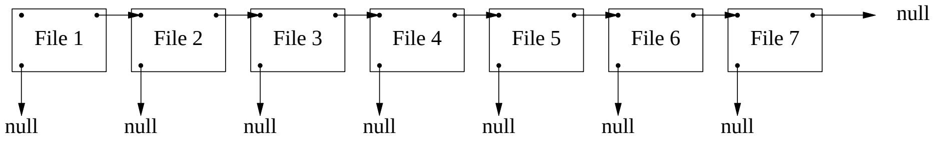
When we compare files, we can not only compare e.g. the first file of the list with all others and throw away all files, which are differing from this file. If we do that, we only has founded files, which are equal to THE FIRST file. We have thrown away files, which maybe are equal, but does not contain the same data as the first one, but maybe all other files are equal.

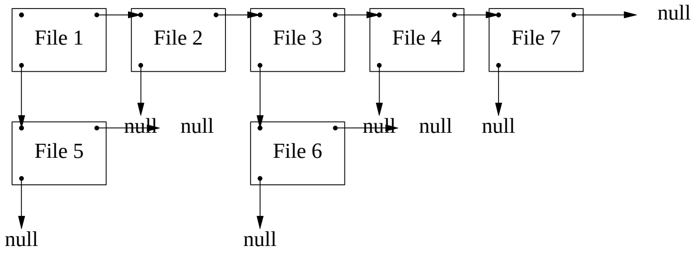
(Thats the last of the two problems, mentioned above.)

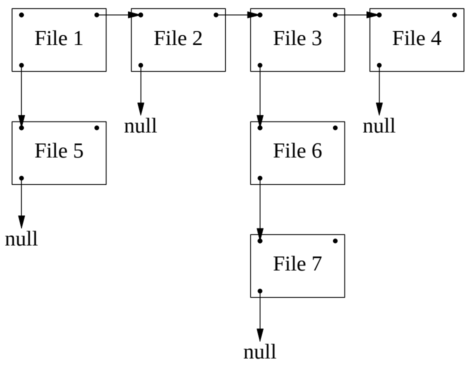
So we maybe think of comparing all files with all others, but that’s not recommended, in respect to performance.

So lets have a look at the algorithm, used here.

We assume for this explanation, that we have seven files. We name these files by number. We assume further, that following files are equal:
file 5 equals file 1
file 6 equals file 3
file 7 equals file 3







```
*/
/* ============================================================== */
/* ================================================================= */
/* ================================================================= */
int compare_same_sized( FILE_CMP** liste )
{
  FILE_CMP* start;
  FILE_CMP* secondfile;
  FILE_CMP* prev_secondfile; /* previous scondfile */
  FILE_CMP* last_clone;
  COUNTER   i;
  int       state;
  int       ret;


  secondfile      = NULL;
  prev_secondfile = NULL;

  i = 0;
  state = START;
  state = A;
  start = *liste;
  last_clone = start;

  while(*(liste+i))
  {

#ifdef TEST
printf("STATE: %c\t", (state - A) + ’A’ );
if( start )
  printf("start: %s\t", start->fname );
if( secondfile )
  printf("secondfile: %s ", secondfile->fname );
printf("\n");
#endif

    switch(state)
    {

      case F: i++;  /* Set Index to next filesize */
              state = A;
              break;


      case A: /* Next Filesize */
              start = *(liste + i);
              secondfile = start;
              last_clone = start;
              state = C;
              break;


      case B: /* Next start-File */

              if( start->next_cmp ) /* more files available ? */
              {
                start = start->next_cmp; /* more files available ? */
                secondfile = start;
                last_clone = start; /* !!! */
                state = C;               /* continue at Point "C" */
                break;
              }
              else
              {
                state = F;
                break;
              }
              break;


      case C: /* next secondfile-File */

              if( secondfile->next_cmp )
              {
                prev_secondfile = secondfile;
                secondfile = secondfile->next_cmp;
                /* NO BREAK!!! */
              }
              else
              {
                state = B;
                break;
              }

              /* comparing two files */
              ret = are_filesequal(start,secondfile);

              if( ret == YES )
                state = D;
              else
                state = C;

              break;



      case D: /* Einsortieren der Files */

              prev_secondfile->next_cmp = secondfile->next_cmp; /* Brücke bauen */
              last_clone->next_clone = secondfile;    /* secondfile unten einhängen */
              last_clone = secondfile;                /* last_clone auf letztes Element setzen */
              last_clone->next_cmp = NULL;            /* last_clone wird nicht mehr verglichen */

              secondfile = prev_secondfile;
              state = C;

              break;
    }
  }
  return 0;
}


/*
```

Function *are_filesequal*

We are comparing two files bytewise here.

```
*/
/* ================================================================ */
/* ================================================================ */
/* ================================================================ */
int are_filesequal(FILE_CMP* a, FILE_CMP* b)
{

  FILE*  fp1;
  FILE*  fp2;

  int    c1;
  int    c2;
  int    equal;


  /* open first file */
  /* --------------- */
  fp1 = fopen(a->fname, "rb");
  if( fp1 == NULL )
  {
    perror("are_filesequal()");
    exit(EXIT_FAILURE);
  }

  /* open second file */
  /* ---------------- */
  fp2 = fopen(b->fname, "rb");
  if( fp2 == NULL )
  {
    fclose(fp1); /* ist ja bereits offen */
    perror("are_filesequal()");
    exit(EXIT_FAILURE);
  }

#ifdef TEST
printf("comparing: %s <-> %s   ", a->fname, b->fname);
#endif

  /* set to startposition of body */
  /* ---------------------------- */
  if( cli_options.ignore_headers )
  {
#ifdef TEST
printf("||| Pos. a: %ld  Pos. b: %ld ||| ", a->cmp_position, b->cmp_position);
#endif
    lseek( fileno(fp1), a->cmp_position, SEEK_SET );
    lseek( fileno(fp2), b->cmp_position, SEEK_SET );
  }


  /* compare files byte by byte */
  /* -------------------------- */
  equal = YES;
  while(1)
  {
    c1 = getc( fp1 );
    c2 = getc( fp2 );

    if( c1 == EOF ) /* Wenn Ende, dann fertig (beide gleichlang!)*/
      break;

    if( c1 != c2 )
    {
      equal = NO;
      break;
    }
  }
  fclose( fp1 );
  fclose( fp2 );

#ifdef TEST
  equal == YES ? printf("=> Files are equal\n") : printf("=> Files are different\n");
#endif

  return equal;
}



/*
```

Function *print_usage*

A hint to the user, how to use multiple.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
void print_usage( FILE* outstream )
{
  fprintf( outstream, "usage: ");
  fprintf( outstream, "find <dirname> | multiple [-a] [-i] [-v] [-h] [-0] [-d]\n");

  return;
}


/*
```

Function *read_until_body*

If we want to ignore all contents of the files, which are before the first empty line (e.g. ignoring mail-/newsheaders), then this function will be invoked to do the job.

The Result is stored in the info-structure (type GET_LENGTH), which will be returned as return-value.

```
*/
/* ------------------------------------------------------------------------ */
/* ------------------------------------------------------------------------ */
/* ------------------------------------------------------------------------ */
GET_LENGTH read_until_body( char* filename )
{
  int    c;
  int    last_c;
  FILE*  file;
  GET_LENGTH  info;

  info.length = 0; /* length of header (startposition of comparision ) */
  info.status = 0;

  /* statt inkremet des Zählers könnte man auch lseek() nutzen! */

  file = fopen(filename, "rb");
  if( file == NULL )
  {
    perror("Error in Function read_until_body()");
    exit(EXIT_FAILURE);
  }

  last_c = 0;
  while( info.status == 0 )
  {
    c = getc( file );

    if( c == EOF )
    {
      info.status = 1; /* end before header detected => seemingly binary file */
      break;
    }
    else
      ++info.length;

    if( c == ’\n’  &&  last_c == ’\n’ )
      break;

    last_c = c;
  }
  fclose(file);

  return(info);
}



/*
```

Function *print_help_message*

The name says all: print_help_message is the name of the game.

```
*/
/* ================================================================= */
/* ================================================================= */
/* ================================================================= */
int print_help_message( FILE* outstream )
{
  fprintf( outstream, "usage: ");
  fprintf( outstream,  "find <dirname> | multiple [-a] [-i] [-v] [-h] [-0] [-d]\n");
  fprintf( outstream, " option a: print ALL filenames instead of superflous.\n");
  fprintf( outstream, " option i: compare after first empty line (IGNORE headers).\n");
  fprintf( outstream, " option v: print version.\n");
  fprintf( outstream, " option h: print this help message.\n");
  fprintf( outstream, " option 0: don’t ignore files with size == 0\n");
  fprintf( outstream, " option d: delete superflous files and print name of the non-deleted files.\n");
  fprintf( outstream, "           Attention! This option ignores read-only-permissions of your files!\n");
  return 0;
}

/* ============= ============= DAS WAR’s  ============= ============= */
/* ============= ============= that’s all ============= ============= */
/* ****************************************************************** */

/*
```