PERSONAL - Cheat Sheet

GRA Cheat Sheet

Using `getopt()`

```
#include <unistd.h> // needed for getopt()
#include <getopt.h> // needed for getopt_long()

// standard getopt, which can only parse single-character command-line
arguments of type "-a" (with opt. argument)
int getopt(int argc, char * const argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

// getopt_long, which allows for multiple-character command-line arguments
of type "--arg"
int getopt_long(int argc, char *const argv[],
                const char *optstring,
                const struct option *longopts, int *longindex);

// necessary for getopt_long:
struct option {
    const char *name; // name of argument as string, e.g. "help" for --help
    int         has_arg; // no_argument, required_argument (if arg req) or
optional_argument
    int        *flag; // if NULL, getopt_long() returns val, otherwise
return 0 and store val at pointer address
    int         val; // a.k.a. shortname (abbrev.); value to return or to
load into variable pointed to by flag address
};

// USEFUL: get program name with "basename(argv[0])" -> #define USAGE_FMT
"%s [-v] [-f hexflag] [-i inputfile] [-o outputfile] [-h]"
```

- **`getopt()` standard definitions**:
  - `argc` (*argument count,* `int`) and `argv` (*argument vector, string array* `char**` *of arguments*) are the **same ones as passed to the `main` function**

    - any element that starts with `-` and is not exactly `-` or `--` is considered an **option element**
        - the characters besides the `-` are **option characters**
    - calling `getopt()` repeatedly successively returns **each of the option characters** from **each of the option elements**
        - this is why `getopt()` should usually be called in a **loop**, where `-1` indicates that no more options are present; see convention below
    - `getopt()` automatically permutes `argv` so that all the non-options are at the end (?)
        - **if and only if** the first character of `optstring` is `+` or `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered
        - if the first character of `optstring` is `-`, then each nonoption `argv`-element is handled as if it were the argument of an option with character code 1
    - the special argument `--` forces an **end of option-scanning** regardless of the scanning mode
- **`getopt()`-specific variables**:
    - `optind`: **index of next element to be processed** in `argv`, initialized to 1 by the system and able to be reset by the caller to restart scanning of an argument vector (*either same one as before or a new one*)
        - once there are no more option characters, `optind` stores the index in `argv` of the first element that is not an option
    - `optstring`: string containing **legitimate option characters** (*one-byte, printable ASCII character that is not space*) that is not `-`, `:` or `;`
        - if a character is followed by a `:`, it **requires** an argument, such that a pointer to the argument (*a.k.a. the next string in `argv`*) is placed in `optarg` (*e.g. if `a:` is included, then for `-a argument`, `optarg` contains the **string (!)** `argument`*)
        - two colons (`::`) indicate that an option element has **an optional argument**
            - if there is text in the **current** `argv` element (*e.g. `-oarg` for `-o`*), then it is returned in `optarg`, otherwise it is set to **zero**
        - if `optstring` contains `W;`, then `-W foo` is treated as the long option `--foo` (*for implementation reasons*)
    - `optopt` contains the **erroneous option character** following an error with return value `?`
    - `opterr` is **set by default** and results in `getopt()` **printing an error message** on standard error
        - if set to 0 in code, no error message gets printed
- **error handling**:

    - *possible errors*: option character found that was **not specified** in `optstring` or **missing option argument**
    - **default (`opterr` set)**: print error message, place erroneous option character in `optopt` and return `?`
    - `opterr == 0`: no error message printed, manual test needed for return value of `?`
    - if the first character of `optstring` is `:`, then the function returns `:` to indicate a missing option argument
- `getopt_long()`:
    - allows **long arguments** (*e.g.* `--help`), specified in `longopts` (*pointer to array of* `struct option`**s**)
        - **(!) the last element of the array has to be filled with zeros (*e.g.* `{NULL, 0, NULL, 0}`)**
    - if `longindex` is not `NULL`, it points to a variable which is set to the index of the long option relative to `longopts`

`getopt()` Conventions

- use `getopt()` in a loop to iterate through each command-line option and terminate loop once the function returns `-1`
- use `switch(c)` to dispatch on the return value from `getopt()`, where for each case, usually, a variable is set that is later used in the program
- use a second loop to process the remaining non-option arguments, starting at `optind` until `argc - 1`

```c
// example code using getopt()
// source: https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>

int main(int argc, char** argv) {
    int aFlag = 0;
    int bFlag = 0;
    char* cValue = NULL;
    int index;
    int c;
    opterr = 0; // handle '?' case
    // a and b don't take any arguments, whereas c does, as indicated by the ':'
    while ((c = getopt(argc, argv, "abc:")) != -1) {
```

```
        switch (c) {
            case 'a':
                aFlag = 1;
                break;
            case 'b':
                bFlag = 1;
                break;
            case 'c':
                cValue = optarg;
                break;
            case '?':
                if (optopt == 'c') fprintf (stderr, "Option -%c requires an
argument.\n", optopt);
                else if (isprint(optopt)) fprintf (stderr, "Unknown option
`-%c'.\n", optopt);
                else fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
                return EXIT_FAILURE;
            default:
                abort();
        }
    }

    printf("aflag = %d, bflag = %d, cvalue = %s\n", aFlag, bFlag, cValue);

    for (index = optind; index < argc; index++) printf ("Non-option argument
%s\n", argv[index]);
    return EXIT_SUCCESS;
}
```

```
// example code using getopt_long()
// source: https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-
Option-Example.html
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

/* Flag set by '--verbose'. */
static int verbose_flag;

int main (int argc, char **argv) {
  int c;
```

```
while (1) {
    static struct option long_options[] =
      {
        /* These options set a flag. */
        {"verbose", no_argument,       &verbose_flag, 1},
        {"brief",   no_argument,       &verbose_flag, 0},
        /* These options don't set a flag.
           We distinguish them by their indices. */
        {"add",     no_argument,       0, 'a'},
        {"append",  no_argument,       0, 'b'},
        {"delete",  required_argument, 0, 'd'},
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0, 'f'},
        {0, 0, 0, 0} // required!
      };

    /* getopt_long stores the option index here. */
    int option_index = 0;
    c = getopt_long (argc, argv, "abc:d:f:", long_options, &option_index);

    /* Detect the end of the options. */
    if (c == -1) break;
    switch (c) {
      case 0: // long option found
        /* If this option set a flag, do nothing else now. */
        if (long_options[option_index].flag != 0) break;
        printf("option %s", long_options[option_index].name);
        if (optarg) printf (" with arg %s", optarg);
        printf("\n");
        break;

      // standard getopt() procedure
      case 'a':
        puts("option -a\n");
        break;

      case 'b':
        puts("option -b\n");
        break;

      case 'c':
        printf("option -c with value `%s'\n", optarg);
        break;
```

```
        case 'd':
          printf("option -d with value `%s'\n", optarg);
          break;

        case 'f':
          printf("option -f with value `%s'\n", optarg);
          break;

        case '?':
          /* getopt_long already printed an error message. */
          break;

        default:
          abort();
        }
    }

  /* Instead of reporting '--verbose'
     and '--brief' as they are encountered,
     we report the final status resulting from them. */
  if (verbose_flag) puts ("verbose flag is set");

  /* Print any remaining command line arguments (not options). */
  if (optind < argc)
    {
      printf ("non-option ARGV-elements: ");
      while (optind < argc) printf ("%s ", argv[optind++]);
      putchar ('\n');
    }

  return EXIT_SUCCESS;
}
```

Converting Strings to Numbers

```
#include <stdlib.h>

// string -> integer (long, long long)
long strtol(const char *restrict nptr, char **restrict endptr, int base);
long long strtoll(const char *restrict nptr, char **restrict endptr, int
base);

// string -> floating point number (float, double, long double)
```

```
double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

- **string to integer**: `strtol()` or `strtoll()`
  - convert initial part of string in `nptr` to a **long integer value** according to given `base` (*between 2 and 36 inclusive or special value 0*) until the first character that is not a valid digit in the given base is reached
  - string may begin with an **arbitrary amount of whitespace** followed by a single optional `+` or `-`
  - if the base is `16` (*i.e. hexadecimal*), string may include optional prefix `0x` or `0X`
  - if the base is `0`:
    - if there is a `0x` or `0X` prefix, the number will be interpreted as a **hex value** (*base 16*)
    - if the first character is `0`, the number will be interpreted as an **octal value** (*base 8*)
    - if neither conditions are met, the number is interpreted as a **decimal number** (*base 10*)
  - for bases **10 to 36**, upper- and lowercase letters represent the same number (*e.g.* `'a' == 'A' == 10`)
  - if `endptr` is not `NULL`, the address of the first invalid character gets stored there
    - if there were no digits, then `endptr = nptr` and the function returns 0
    - *formally*: if `*nptr != '\0' && **endptr == '\0'`, then the **entire string** was **valid**
  - **return value**:
    - *standard*: string parsed as `long`
    - *over- or underflow*: `(L)LONG_MAX` (*OF*) / `(L)LONG_MIN` (*UF*) and `errno = ERANGE` (*both cases*)
      - since both values can be legitimate numbers, the code should set `errno = 0` before the function call and check if `errno != 0` afterwards to see if an actual over- or underflow occured
    - *invalid base*: `errno = EINVAL`
- **string to floating point number**: `strtod()` (*or the other ones but who cares about those*)
  - convert initial part of string in `nptr` to a **double**
  - string may begin with an **arbitrary amount of whitespace** followed by a single optional `+` or `-` and then either…
    - …a decimal number (*non-empty sequence of decimal digits possibly containing a locale-dependent radix character, most often* `.`*, optionally followed by a decimal exponent* `E` *or* `e`*, followed by another optional* `+` *or* `-`*, followed by another non-empty sequence of decimal digits to indicate multiplication by a power of 10*)

        - …a hexadecimal number (`0x` or `0X` *followed by a non-empty sequence of hexadecimal digits*)
        - …`infinity` / `inf` (*case-insensitive*)
        - …`NaN` (*case-insensitive*)
    - if `endptr` is not `NULL`, the address of the first invalid character gets stored there
        - if there were no digits, then `endptr = nptr` and the function returns `0.0`
        - *formally*: if `*nptr != '\0' && **endptr == '\0'`, then the **entire string** was **valid**
    - **return value**:
        - *standard*: string parsed as `double`
        - *overflow*: return plus or minus `HUGE_VAL` and set `errno = ERANGE`
        - *underflow*: a value with magnitude no larger than `DBL_MIN` is returned and `errno = ERANGE`

```c
// example for strtol()
// source: man 3 strtol
#include <stdlib.h>
#include <limits.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
   int base;
   char *endptr, *str;
   long val;

   if (argc < 2) {
       fprintf(stderr, "Usage: %s str [base]\n", argv[0]);
       exit(EXIT_FAILURE);
   }

   str = argv[1];
   base = (argc > 2) ? atoi(argv[2]) : 0; // ew, atoi

   errno = 0;    /* To distinguish success/failure after call */
   val = strtol(str, &endptr, base);

   /* Check for various possible errors. */

   if (errno != 0) {
```

```
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    if (endptr == str) {
        fprintf(stderr, "No digits were found\n");
        exit(EXIT_FAILURE);
    }

    /* If we got here, strtol() successfully parsed a number. */

    printf("strtol() returned %ld\n", val);

    if (*endptr != '\0')        /* Not necessarily an error... */
        printf("Further characters after number: \"%s\"\n", endptr);

    exit(EXIT_SUCCESS);
}
```

File I/O

- **opening a file** and associating a **stream** with it: `FILE* fopen(const char* restrict pathname, const char* restrict mode)`
  - `pathname`: string that contains the **path** to the file
  - `mode`: string beginning with one of the following sequences:
    - `r`: open file for **reading**, starting at beginning of file
    - `r+`: open file for **reading and writing**, starting at beginning of file
    - `w`: open and **truncate to zero length / create file** for **writing**, starting at beginning of file
    - `w+`: open and **truncate to zero length / create file** for **reading and writing**, starting at beginning of file
    - `a`: open for **appending / create file for writing**, starting at the end of the file
    - `a+`: open for **reading and appending / create file** for **reading and writing**, starting at the end of the file
  - **return value**: `FILE` pointer if successful, otherwise `NULL` and `errno` is set
- **accessing file metadata**: `int fstat(int fd, struct stat *sb)`
  - **(!) file must be converted to file descriptor using `fileno(file)`**
  - metadata is written to an object of type `struct stat` (*e.g.* `struct stat statbuf; fstat(fileno(file), &statbuf)`)

        - `st_dev`: numeric ID of the device containing the file
        - `st_ino`: the file's inode number
            - `st_dev` and `st_ino` are enough to uniquely identify a file within the system
        - `st_nlink`: number of hard links to the file
        - `st_flags`: flags enabled for the file, see `chflags`
        - `st_atim`: time when file data was last accessed
        - `st_mtim`: time when file data was last modified
        - `st_ctim`: time when file data was last changed (*inode data modification*)
        - `st_birthtim`: time when inode was created
        - `st_size`: file size in bytes
        - `st_blksize`: optimal I/O block size for file
        - `st_blocks`: actual number of blocks allocated for file in 512-byte units
        - `st_uid`: user ID of file's owner
        - `st_gid`: group ID of file
        - `st_mode`: status of file - can be tested using macros (*0 if false, non-zero if true*)
            - `S_ISBLK(m)`: test for a block special file
            - `S_ISCHR(m)`: test for a character special file
            - `S_ISDIR(m)`: test for a directory
            - `S_ISFIFO(m)`: test for a pipe or FIFO special file
            - `S_ISLNK(m)`: test for a symbolic link
            - `S_ISREG(m)`: test for a regular file
            - `S_ISSOCK(m)`: test for a socket
            - `S_ISWHT(m)`: test for a whiteout
    - **return value**: 0 on success, -1 otherwise and `errno` set
- **reading a file**: `size_t fread(void* restrict ptr, size_t size, size_t nmemb, FILE* restrict stream)`
    - read `nmemb` bytes of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at memory address `ptr`
    - **return value**: if successful, number of items read (*equal to number of bytes transferred if `size == 1`*); otherwise, short item count or zero (*e.g. incomplete value*)
    - **(!) `fread()` does not distinguish between EoF and error, `feof()` and `ferror()` must be use to determine cause of error**

- **writing to a file**: `size_t fwrite(const void* restrict ptr, size_t size, size_t nmemb, FILE* restrict stream)`
  - write `nmemb` bytes of data, each `size` bytes long, to the stream pointed to by `stream`, reading bytes from memory address `ptr`
  - **return value**: if successful, number of items written (*equal to number of bytes transferred if `size == 1`*); otherwise, short item count or zero (*e.g. incomplete value*)
- **(!) closing a file**: `fclose(FILE* fp)`

```c
// example code
// source: ERA W07

// read contents of file to string
static char* read_file(const char* path) {
    char* string = NULL;
    FILE* file;

    // check if file can be opened with reading permissions
    if (!(file = fopen(path, "r"))) {
        perror("Error opening file!");
        return NULL;
    }

    // check if file statistics can be retrieved and stored in statbuf
    struct stat statbuf;
    if (fstat(fileno(file), &statbuf)) {
        fprintf(stderr, "Error retrieving file stats!\n");
        goto cleanup;
    }

    // check if file is regular and has valid size
    if (!S_ISREG(statbuf.st_mode) || statbuf.st_size <= 0) {
        fprintf(stderr, "Error processing file: Not a regular file or
invalid size!\n");
        goto cleanup;
    }

    // allocate sufficient bytes for file contents and extra null-byte to
terminate string
    if (!(string = malloc(statbuf.st_size + 1))) {
        fprintf(stderr, "Error reading file: Could not allocate enough
memory!\n");
        goto cleanup;
```

```
    }

    // read contents and check if number of bytes read matches with file
size in bytes
    if (fread(string, 1, statbuf.st_size, file) != (size_t) statbuf.st_size)
{
        fprintf(stderr, "Error reading file!\n");
        free(string);
        string = NULL;
        goto cleanup;
    }

    // append null byte
    string[statbuf.st_size] = '\0';

    // perform end-of-function / failure cleanup to prevent memory leak (!)
    cleanup:
    if (file) fclose(file);

    // returns the string containing the data from the file or NULL if
something went wrong
    return string;
}

// write string to file
static void write_file(const char* path, const char* string) {
    FILE* file;

    // check if file can be opened with writing permissions
    if (!(file = fopen(path, "w"))) {
        perror("Error opening file!");
        return;
    }

    // attempt to write contents and check if number of bytes written
matches with file size in bytes
    const size_t stringlength = strlen(string);
    if (fwrite(string, 1, stringlen, file) != stringlen) {
        fprintf(stderr, "Error writing to file!\n");
    }

    // once again, close the file when done!
```

```
    fclose(file);
}
```

```
// some more example code
// source: wikipedia
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char buffer[5];
    FILE* fp = fopen("myfile", "rb");

    if (fp == NULL) {
        perror("Failed to open file \"myfile\"");
        return EXIT_FAILURE;
    }

    for (int i = 0; i < 5; i++) {
        int rc = getc(fp);
        if (rc == EOF) {
            fputs("An error occurred while reading the file.\n", stderr);
            return EXIT_FAILURE;
        }

        buffer[i] = rc;
    }

    fclose(fp);

    printf("The bytes read were... %02x %02x %02x %02x %02x\n", buffer[0],
buffer[1], buffer[2], buffer[3], buffer[4]);

    return EXIT_SUCCESS;
}
```

Compiler Options

- **set standard to C17**: `-std=c17`
- **output debugging information (!)**: `-g`
- **generate pre-processor source code**: `gcc -o file.i -E file.c`
- **compile to ASM file using Intel syntax**: `gcc -o file.S -S file.c -masm=intel`

- instrument the branches and calls in the program and create a **coverage notes** file for `gcov`: `--coverage`
- **warning flags**:
  - `-Wextra -Wall`: essential
  - `-Wfloat-equal`: useful because usually testing floating-point numbers for equality is bad
  - `-Wundef`: warn if an uninitialized identifier is evaluated in an `#if` directive
  - `-Wshadow`: warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed
  - `-Wpointer-arith`: warn if anything depends upon the size of a function or of `void`
  - `-Wcast-align`: warn whenever a pointer is cast such that the required alignment of the target is increased (*for example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries*)
  - `-Wstrict-prototypes`: warn if a function is declared or defined without specifying the argument types
  - `-Wstrict-overflow=5`: warns about cases where the compiler optimizes based on the assumption that signed overflow does not occur (*5 may be too strict*)
  - `-Wwrite-strings`: give string constants the type `const char[length]` so that copying the address of one into a non-`const char *` pointer will get a warning
  - `-Waggregate-return`: warn if any functions that return structures or unions are defined or called
  - `-Wcast-qual`: warn whenever a pointer is cast to remove a type qualifier from the target type (?)
  - `-Wswitch-default`: warn whenever a `switch` statement does not have a `default` case (?)
  - `-Wswitch-enum`: warn whenever a `switch` statement has an index of enumerated type and lacks a `case` for one or more of the named codes of that enumeration (?)
  - `-Wconversion`: warn for implicit conversions that may alter a value (?)
  - `-Wunreachable-code`: warn if the compiler detects that code will never be executed (?)

Debugging with GDB

- **important**: program must be compiled using `-g` to generate debug info!
- **useful commands**:
  - **display contents of all registers**: `info all-registers`
  - **display contents of a specific register (*ideally, xmm-registers*)**: `p $reg`
  - **print value of variable in decimal**: `p name`
  - **print what is pointed to by address stored in variable**: `p * name`

    - **print value of variable in hex**: `p/x name`
- **debugging**:
    1. start GDB (`gdb ./file`)
    2. set breakpoint for `main` function in C (`b main`) (*alternatively: set breakpoint for specific function, `main` is universal*)
    3. `layout regs` shows the values loaded in each register at the top, the source file (*C / ASM*) in the middle and the terminal at the bottom
    4. `r` runs the program
    5. `s` runs the next line of the program

```
$ gdb ./main
$ b main
$ layout regs
$ r
$ s
```

ASM File Layout

```
.intel_syntax noprefix  // ensure intel syntax because intel rules and AT&T drools
.global function_name   // make function externally visible to C program

.text
function_name:
    ...
    ret
```

Makefile Layout

- first target in `makefile` is the default target
- `$@`: name of **target**
- `$^`: names of **all** prerequisites
- `$<`: name of **first** prerequisite

```
# Standard Rule layout

# Target contains prerequisites "prerequisite1", "prerequisite2", ...
# These prerequisites can be source code files or other targets.
# The recipes are shell commands.

target : prerequisite1 prerequisite2 ...

```

```
    recipe1
    recipe2
```

```
# Layout of a typical Makefile to compile any C program
# Source: https://opensource.com/article/18/8/what-how-makefile

# Usage:
# make                          # compile all binary
# make clean                    # remove ALL binaries and objects

.PHONY = all clean              # define phony targets all and clean

CC = gcc                        # compiler to use

LINKERFLAG = -lm                # define flags to be used with GCC in a
recipe

SRCS := $(wildcard *.c)         # function for filenames: stores all files
with extension .c
BINS := $(SRCS:%.c=%)           # substitution reference: output files have
same names as source files, without .c extension

# phony target all calls values in${BINS} as individual targets
all: ${BINS}

# Equivalent to:
# foo: foo.o
#       @echo "Checking.."
#       gcc -lm foo.o -o foo

%: %.o
    @echo "Checking.."
    ${CC} ${LINKERFLAG} $< -o $@

# Equivalent to:
# foo.o: foo.c
#       @echo "Creating object.."
#       gcc -c foo.c

%.o: %.c
    @echo "Creating object.."
    ${CC} -c $<
```

```
# remove binaries and object files
clean:
    @echo "Cleaning up..."
    rm -rvf *.o ${BINS}
```

```
# Rewritten, assuming only one file (foo.c) is used

.PHONY = all clean

CC = gcc

LINKERFLAG = -lm

SRCS := foo.c
BINS := foo

all: foo

foo: foo.o
    @echo "Checking.."
    gcc -lm foo.o -o foo

foo.o: foo.c
    @echo "Creating object.."
    gcc -c foo.c

clean:
    @echo "Cleaning up..."
    rm -rvf foo.o foo
```

Summary by Flavius Schmidt, ge83pux, 2023.
https://home.in.tum.de/~scfl