1. Introduction. This is the source for assignment 060. Here's some instructions from README:

> In this assignment, you will be producing your own, simplified version of the UNIX utility `stat`, which provides information about a file. ... Your ultimate goal is to write a program called `mystat` which replicates the functionality of the `stat` command without any options (that is, you aren't going to do `-t`, `-c`, etc). If you run `stat something` and `mystat something` they should produce exactly the same output.

2. Below are some header files already included in `mystat.c`. I may want to include more when needed.

```
⟨ headers 2 ⟩ ≡
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <pwd.h>
#include <grp.h>
#include <unistd.h>
```

See also section 6.

This code is used in section 4.

3. And a *time2str* function is provided, too.

```
⟨ definitions 3 ⟩ ≡        /* This function is for Step 4 */
  char *time2str(const time_t *when, long ns)
  {
    char *ans = malloc(128 * sizeof (*ans));
    char temp1[64];
    char temp2[32];
    const struct tm *t = localtime(when);
    strftime(temp1, 512, "%Y-%m-%d %H:%M:%S", t);
    strftime(temp2, 32, "%z", t);
    snprintf(ans, 128, "%s.%09ld %s", temp1, ns, temp2);
    return ans;
  }
```

See also sections 7, 15, 18, and 25.

This code is used in section 4.

4. The whole program will look like this:

```
⟨ headers 2 ⟩
⟨ declarations 14 ⟩
⟨ definitions 3 ⟩
⟨ main program 5* ⟩
```

5* Step 1. As in README:

> Your first goal is to make your mystat program accept exactly ONE filename as a command line argument, and print out the first THREE lines of the output that `stat` would print for that file.

In Step 1, I wrote a main function that only needs to process *argv*[1].

In Step 5, README says

> Up to this point, your program has taken ONE command line argument, however, the real stat program takes an arbitrary number of arguments, and stats each one. For example, you can run `stat README . /dev/null` and it will print the information for all three files one after the other.

So finally the *main* function here is able to handle multiple arguments.

```
⟨ main program 5* ⟩ ≡
  int main(int argc, char **argv)
  {
    int i;
    int rc = EXIT_SUCCESS;
    if (argc < 2) {      /* stat is run without arguments */
      fprintf(stderr,
          "stat: missing operand\n"
          "Try 'stat --help' for more information.\n");
      return EXIT_FAILURE;
    }
    for (i = 1; i < argc; i++) {
      if (printstat(argv[i]) ≠ 0) {
        rc = EXIT_FAILURE;      /* stat returns failure if any argument caused an error */
      }
    }
    return rc;
  }
```

This code is used in section 4.

6. Before proceeding, I think there are some headers very useful.

```
⟨ headers 2 ⟩ +≡
#include <assert.h>     /* assert */
#include <string.h>     /* strcpy, strerror, etc. */
#include <errno.h>      /* errno */
```

7. *printstat* is the `stat` program for a single file. For example, passing `"README"` to it might produce output like this:

```
File: ‘README’
Size: 2825          Blocks: 8           IO Block: 4096       regular file
Device: fc00h/64512d Inode: 801829       Links: 1
```

Note The README said I should use the fancy quotes (‘’). However, this specification is later changed (see Piazza @366), and I should use ordinary quotes (`''`) instead.

```
⟨ definitions 3 ⟩ +≡
  int printstat(const char *filename)
  {
    struct stat st;
    ⟨ read stats into st 8 ⟩
    ⟨ print formatted stats 9* ⟩
    return 0;      /* no error happened */
  }
```

8. I can make use of the *lstat* system call to read the stats into a **struct stat**.

Before printing anything, I need to check if the call succeeded. (Things like nonexistent file or permission denied might happen.)

```
⟨ read stats into st 8 ⟩ ≡
  if (lstat(filename, &st) ≠ 0) {      /* lstat returns non-zero on error */
    fprintf(stderr, "stat: cannot stat '%s': %s\n", filename, strerror(errno));
    return errno;      /* don't call exit; real stat program proceeds and deals with other files */
  }
```

This code is used in section 7.

9* On the first line I only need to print the filename.

But (in step 7) when the file is a symbolic link, I should also output what it links to.

```
⟨ print formatted stats 9* ⟩ ≡
  if (S_ISLNK(st.st_mode)) {      /* file is a symbolic link */
    char buf[256];
    ssize_t len;
    len = readlink(filename, buf, 256);
    assert(len ≥ 0);      /* negative length does not make sense */
    if (len > 255) {
      buf[255] = '\0';
    }
    else {
      buf[len] = '\0';
    }
    printf("  File: '%s' -> '%s'\n", filename, buf);
  }
  else {
    printf("  File: '%s'\n", filename);
  }
```

See also sections 10, 11*, 12, and 23.

This code is used in section 7.

10. On the second line I need to print ‘Size’, ‘Blocks’, and ‘IO Block’.

```
⟨ print formatted stats 9* ⟩ +≡
  {
    const char *filetype = Λ;
    switch (st.st_mode & S_IFMT) {
    case S_IFBLK: filetype = "block special file"; break;
    case S_IFCHR: filetype = "character special file"; break;
    case S_IFDIR: filetype = "directory"; break;
    case S_IFIFO: filetype = "fifo"; break;
    case S_IFLNK: filetype = "symbolic link"; break;
    case S_IFREG: filetype = "regular file"; break;
    case S_IFSOCK: filetype = "socket"; break;
    }
    assert(filetype ≠ Λ);      /* none of the possibilities described in README! I'm confused... */
    printf("  Size: %-10lu\tBlocks: %-10lu IO Block: %-6lu %s\n",
        st.st_size, st.st_blocks, st.st_blksize, filetype);
  }
```

11*. On the third line, I need to print ‘Device’, ‘Inode’, and ‘Links’.

Additionally (in step 6), another field called “Device type” will be appended to the third line if the file happens to be a device.

```
⟨ print formatted stats 9* ⟩ +≡
  printf("Device: %lxh/%lud\tInode: %-10lu", st.st_dev, st.st_dev, st.st_ino);
  if (S_ISCHR(st.st_mode) ∨ S_ISBLK(st.st_mode)) {      /* file is a device */
    printf("  Links: %-5lu Device type: %d,%d\n",
        st.st_nlink, major(st.st_rdev), minor(st.st_rdev));
  }
  else {
    printf("  Links: %lu\n", st.st_nlink);
  }
```

12. Step 2. Now I need to deal with the fourth line in the formatted output. To leave room for future extension, I will not hard-code what to print here.

```
⟨ print formatted stats 9* ⟩ +≡
  ⟨ print the fields in the fourth formatted line 13 ⟩
  printf("\n");      /* terminate the fourth line */
```

13. At this point, only the first field ("Access") needs to be output on the fourth line. It consists of two parts. The first is an octal code of the permissions. I need to mask *st_mode* with ∼S_IFMT to get that code. The second part is a human-readable description (I will call it "mode string" in later descriptions), which can be generated by *get_modestr*.

```
#define MODESTR_LEN 10      /* length of the mode string, not including '\0' */
⟨ print the fields in the fourth formatted line 13 ⟩ ≡
  {
    char modestr[MODESTR_LEN + 1];      /* +1 for '\0' */
    printf("Access: %04o/%s", st.st_mode & ∼S_IFMT, get_modestr(modestr, st.st_mode));
  }
```

See also sections 21 and 22.

This code is used in section 12.

14. *get_modestr* generates the mode string.

```
⟨ declarations 14 ⟩ ≡
  char *get_modestr(char *buf, int mode);
```

See also sections 17 and 24.

This code is used in section 4.

15.

```
⟨ definitions 3 ⟩ +≡
  char *get_modestr(char *buf, int mode)
  {
    ⟨ write file type into buf[0] 16 ⟩
    ⟨ write permissions into buf[1]...buf[MODESTR_LEN − 1] 19 ⟩
    buf[MODESTR_LEN] = '\0';      /* terminate the string */
    return buf;
  }
```

16. The first character in the mode string will need a big switch similar to the one in Step 1. But some characters are different, so I cannot reuse the code.

```
⟨ write file type into buf[0] 16 ⟩ ≡
  {
    char c = 0;
    switch (mode & S_IFMT) {
    case S_IFBLK: c = 'b'; break;
    case S_IFCHR: c = 'c'; break;
    case S_IFDIR: c = 'd'; break;
    case S_IFIFO: c = 'p'; break;
    case S_IFLNK: c = 'l'; break;
    case S_IFREG: c = '-'; break;
    case S_IFSOCK: c = 's'; break;
    }
    assert(c ≠ 0);      /* Again, I'm confused when run out of possibilities. */
    buf[0] = c;      /* write it into buf[0] */
  }
```

This code is used in section 15.

17. *get_permissions* unifies the formatting of user, group and other permissions.

```
⟨ declarations 14 ⟩ +≡
  void get_permissions(char *buf, int mode, int rmask, int wmask, int xmask);
```

18.

```
⟨ definitions 3 ⟩ +≡
  void get_permissions(char *buf, int mode, int rmask, int wmask, int xmask)
  {
    buf[0] = (mode & rmask) ? 'r' : '-';
    buf[1] = (mode & wmask) ? 'w' : '-';
    buf[2] = (mode & xmask) ? 'x' : '-';
  }
```

19. Now I can call *get_permissions* to fill in the corresponding fields in the mode string.

```
⟨ write permissions into buf[1]... buf[MODESTR_LEN − 1] 19 ⟩ ≡
  get_permissions(buf + 1, mode, S_IRUSR, S_IWUSR, S_IXUSR);
  get_permissions(buf + 4, mode, S_IRGRP, S_IWGRP, S_IXGRP);
  get_permissions(buf + 7, mode, S_IROTH, S_IWOTH, S_IXOTH);
```

This code is used in section 15.

20. Step 3. Now deal with the other fields in the fourth line. They are "Uid" and "Gid".

21. First deal with "Uid". I can use the *getpwuid* library function to look up the user's name from it's user ID.

Be careful about the leading spaces in the output. They are here to separate the current field from the previous one.

```
⟨ print the fields in the fourth formatted line 13 ⟩ +≡
  {
    struct passwd *pw;
    pw = getpwuid(st.st_uid);
    printf("␣␣Uid:␣(%5d/%8s)", st.st_uid, pw→pw_name);
  }
```

22. Then "Gid". This time use *getgrgid*.

```
⟨ print the fields in the fourth formatted line 13 ⟩ +≡
  {
    struct group *gr;
    gr = getgrgid(st.st_gid);
    printf("␣␣␣Gid:␣(%5d/%8s)", st.st_gid, gr→gr_name);
  }
```

23. Step 4. In this step I will add four more lines to the output. They are the last ones, and are related to time information.

Again, these lines have some similarities, so I will put the actual printing code into a separate function *print_timeinfo*.

```
⟨ print formatted stats 9* ⟩ +≡
  print_timeinfo("Access", &st.st_atim);
  print_timeinfo("Modify", &st.st_mtim);
  print_timeinfo("Change", &st.st_ctim);
  printf("␣Birth:␣-\n");      /* "Birth" line is always this */
```

24.
```
⟨ declarations 14 ⟩ +≡
  void print_timeinfo(const char *what, const struct timespec *when);
```

25. *print_timeinfo* uses the provided *time2str* function to format the time value into a string.

```
⟨ definitions 3 ⟩ +≡
  void print_timeinfo(const char *what, const struct timespec *when)
  {
    char *timestr;
    timestr = time2str(&when→tv_sec, when→tv_nsec);
    printf("%s:␣%s\n", what, timestr);
    free(timestr);
  }
```

26. Step 5, 6, 7. These steps require modifications to previous steps. The modifications are already in place. So no code will be shown here. An asterisk next to a section number indicates modification.

27* Index. Here is the index of all C identifiers, in alphabetical order. The listed numbers are section numbers. The underlined ones are where the corresponding identifiers are defined.

The following sections were changed by the change file: 5, 9, 11, 27.

⟨ declarations 14, 17, 24 ⟩ Used in section 4.
⟨ definitions 3, 7, 15, 18, 25 ⟩ Used in section 4.
⟨ headers 2, 6 ⟩ Used in section 4.
⟨ main program 5* ⟩ Used in section 4.
⟨ print formatted stats 9*, 10, 11*, 12, 23 ⟩ Used in section 7.
⟨ print the fields in the fourth formatted line 13, 21, 22 ⟩ Used in section 12.
⟨ read stats into *st* 8 ⟩ Used in section 7.
⟨ write file type into *buf*[0] 16 ⟩ Used in section 15.
⟨ write permissions into *buf*[1]... *buf*[MODESTR_LEN − 1] 19 ⟩ Used in section 15.

MYSTAT