System-calls and system-defined structures

Most system-calls accept integers and pointers to characters as parameters, and typically return integer values indicating their success. When more information must be passed to a system-call, or the call needs to return multiple values, we employ system-defined (C11) structures defined in operating system provided header files.

Accessing structures using pointers

We've seen that we can access fields of a structure using a single dot ('.' or fullstop).
What if, instead of accessing the structure directly, we only have a *pointer to a structure*?

We've seen "one side" of this situation, already - when we passed the address of a structure to a function:

```
struct timeval   start_time;

gettimeofday( &start_time, NULL );
```

The function *gettimeofday()*, must have been declared to receive a pointer:

```
extern int gettimeofday( struct timeval *time, ......);
```

Consider the following example, in which *a pointer to a structure* is *returned* from a function.
We now use the → operator (pronounced the 'arrow', or 'points-to' operator) to access the fields via the pointer:

```
#include  <stdio.h>
#include  <time.h>

void greeting(void)
{
    time_t      NOW     = time(NULL);
    struct tm   *tm     = localtime(&NOW);

    printf("Today's date is %i/%i/%i\n",
             tm->tm_mday, tm->tm_mon + 1, tm->tm_year + 1900);

    if(tm->tm_hour < 12) {
        printf("Good morning\n");
    }
    else if(tm->tm_hour < 17) {
        printf("Good afternoon\n");
    }
    else {
        printf("Good evening\n");
    }
}
```

Another example - accessing a system's password entries

On a stand-alone Linux system, one not dependent on a network-based server to provide its user information, some local user information is (historically) stored in the textfile */etc/passwd*, with each user's information stored one-per-line with fields separated by colons.

Rather than expecting every user program to parse this information correctly, Linux systems host standard *XOPEN* header files and libraries to conveniently provide the information.

We can iterate through this information with the help of the *getpwent()* function, which returns a *pointer to a 'struct passwd' structure*:

```
#include  <stdio.h>
#define __USE_XOPEN_EXTENDED
#include <sys/types.h>
#include <pwd.h>

#if 0
The header file <pwd.h> declares:

   struct passwd {
       char   *pw_name;       /* username */
       char   *pw_passwd;     /* user password */
       uid_t   pw_uid;        /* user ID */
       gid_t   pw_gid;        /* group ID */
       char   *pw_gecos;      /* user information */
       char   *pw_dir;        /* home directory */
       char   *pw_shell;      /* shell program */
   };

   struct passwd *getpwent(void);
#endif

int main(int argc, char *argv[])
{
    struct passwd *pwd;

    while((pwd = getpwent()) != NULL) {
        printf("%20s\t%-30s\t%s\n",
                pwd->pw_name, pwd->pw_dir, pwd->pw_shell);
    }
    return 0;
}
```

Defining our own datatypes

We can further simplify our code, and more clearly identify related data by defining *our own datatypes*.

Recall material from Lecture-6 where structures, and arrays of structures were introduced. We preceded each structure's name with the **struct** keyword, and accessed structure elements using the 'dot' operator.

Instead, we can use the **typedef** keyword to define our own *new* datatype in terms of an *old* (existing) datatype, and then not have to always provide the **struct** keyword:

```
//  DEFINE THE LIMITS ON PROGRAM'S DATA-STRUCTURES
#define MAX_TEAMS               24
#define MAX_TEAMNAME_LEN        30
....

typedef struct {
    char    teamname[MAX_TEAMNAME_LEN+1];       // +1 for null-byte
    ....
    int     played;
    ....
} TEAM;

TEAM    team[MAX_TEAMS];
```

As a convention (but not a C11 requirement), we'll define our user-defined types using uppercase names:

```
//  PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<nteams ; ++t) {
    TEAM    *tp = &team[t];

    printf("%s %i %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
            tp->teamname,
            tp->played, tp->won, tp->lost, tp->drawn,
            tp->bfor, tp->bagainst,
            (100.0 * tp->bfor / tp->bagainst),      // calculate percentage
            tp->points);
}
```

Defining our own datatypes, *continued*

Let's consider another example - the starting (home) and ending (destination) bustops from the CITS2002 1st project of 2015.
We starting with some of its definitions:

```
//  GLOBAL CONSTANTS, BEST DEFINED ONCE NEAR THE TOP OF FILE
#define MAX_FIELD_LEN                   100
#define MAX_STOPS_NEAR_ANYWHERE         200     // in Transperth:  184

....

//  2-D ARRAY OF VIABLE STOPS FOR COMMENCEMENT OF JOURNEY
char    viable_home_stopid [MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
char    viable_home_name   [MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
int     viable_home_metres [MAX_STOPS_NEAR_ANYWHERE];
int     n_viable_homes  = 0;

//  2-D ARRAY OF VIABLE STOPS FOR END OF JOURNEY
char    viable_dest_stopid [MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
char    viable_dest_name   [MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
int     viable_dest_metres [MAX_STOPS_NEAR_ANYWHERE];
int     n_viable_dests  = 0;
```

(After a post-project workshop) we later modified the 2-dimensional arrays to use dynamically-allocated memory and 'pointers-to-pointers':

```
//  2-D ARRAY OF VIABLE STOPS FOR COMMENCEMENT OF JOURNEY
char    **viable_home_stopid            = NULL;
char    **viable_home_name              = NULL;
int     *viable_home_metres             = NULL;
int     n_viable_homes                  = 0;

//  2-D ARRAY OF VIABLE STOPS FOR END OF JOURNEY
char    **viable_dest_stopid            = NULL;
char    **viable_dest_name              = NULL;
int     *viable_dest_metres             = NULL;
int     n_viable_dests                  = 0;
```

We can now gather the (many) related global variables into a single structure, and use **typedef** to define our own datatype:

```
//  A NEW DATATYPE TO STORE 1 VIABLE STOP
typedef struct {
    char    *stopid;
    char    *name;
```

```
    int     metres;
} VIABLE;

//  A VECTOR FOR EACH OF THE VIABLE home AND dest STOPS
VIABLE   *home_stops         = NULL;
VIABLE   *dest_stops         = NULL;

int      n_home_stops        = 0;
int      n_dest_stops        = 0;
```

Finding the attributes of a file

As seen in Lecture-15, many operating systems manage their data in a file system, in particular maintaining files in a hierarchical directory structure - directories contain files and other (sub)directories.

As we saw with time-based information, we may request file- and directory information from the operating system by calling system-calls. We may employ another POSIX† function, *stat()*, and the system-provided structure *struct stat*, to determine the *attributes* of each file:

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

char *progname;

void file_attributes(char *filename)
{
  struct stat  stat_buffer;

  if(stat(filename, &stat_buffer) != 0) {  // can we 'stat' the file's attributes?
    perror( progname );
    exit(EXIT_FAILURE);
  }
  else if( S_ISREG( stat_buffer.st_mode ) ) {
    printf( "%s is a regular file\n", filename );
    printf( "is %i bytes long\n", (int)stat_buffer.st_size );
    printf( "and was last modified on %i\n", (int)stat_buffer.st_mtime);

    printf( "which was %s", ctime( &stat_buffer.st_mtime) );
  }
}
```

†*POSIX is an acronym for "Portable Operating System Interface", a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix (such as macOS and Linux) and other operating systems (e.g. Windows has a POSIX emulation layer).*

Reading the contents of a directory

Most modern operating systems store their data in *hierarchical* file systems, consisting of directories which hold items that, themselves, may either be files or directories.

The formats used to store information in directories in different file-systems are different(!), and so when writing portable C programs, we prefer to use functions that work portably.

Consider the strong similarities between opening and reading a (text) file, and opening and reading a directory:

```c
#include  <stdio.h>


void print_file(char *filename)
{
    FILE  *fp;
    char  line[BUFSIZ];

    fp       = fopen(filename, "r");
    if(fp == NULL) {
        perror( progname );
        exit(EXIT_FAILURE);
    }

    while(fgets(line, sizeof(buf), fp) != NULL) {
        printf( "%s", line);
    }
    fclose(fp);
}
```

```c
#include  <stdio.h>
#include  <sys/types.h>
#include  <dirent.h>

void list_directory(char *dirname)
{
    DIR             *dirp;
    struct dirent   *dp;

    dirp       = opendir(dirname);
    if(dirp == NULL) {
        perror( progname );
        exit(EXIT_FAILURE);
    }

    while((dp = readdir(dirp)) != NULL) {
        printf( "%s\n", dp->d_name );
    }
    closedir(dirp);
}
```

With directories, we're again discussing functions that are not part of the C11 standard, but are defined by POSIX standards.

The inconsistent naming of system-defined datatypes - for example, *struct dirent* versus *DIR* - can be confusing (annoying) but, over time, renaming datatypes across billions of lines of open-source code becomes impossible.

Investigating the contents of a directory

We now know how to open a directory for reading, and to determine the names of all items in that directory.

What is each "thing" found in the directory - is it a directory, is it a file...?

To answer those questions, we need to employ the POSIX function, *stat()*, to determine the *attributes* of the items we find in directories:

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <sys/param.h>
#include  <dirent.h>
#include  <unistd.h>

void list_directory(char *dirname)
{
    char  fullpath[MAXPATHLEN];

    .....
    while((dp = readdir(dirp)) != NULL) {
        struct stat  stat_buffer;


        sprintf(fullpath, "%s/%s", dirname, dp->d_name );

        if(stat(fullpath, &stat_buffer) != 0) {
             perror( progname );
        }
        else if( S_ISDIR( stat_buffer.st_mode )) {
            printf( "%s is a directory\n", fullpath );
        }
        else if( S_ISREG( stat_buffer.st_mode )) {
            printf( "%s is a regular file\n", fullpath );
        }
        else {
            printf( "%s is unknown!\n", fullpath );
        }
    }
    closedir(dirp);
}
```

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <sys/param.h>
#include  <dirent.h>
#include  <unistd.h>

void list_directory(char *dirname)
{
    char  fullpath[MAXPATHLEN];

    .....
    while((dp = readdir(dirp)) != NULL) {
        struct stat  stat_buffer;
        struct stat  *pointer = &stat_buffer;

        sprintf(fullpath, "%s/%s", dirname, dp->d_name );

        if(stat(fullpath, pointer) != 0) {
             perror( progname );
        }
        else if( S_ISDIR( pointer->st_mode )) {
            printf( "%s is a directory\n", fullpath );
        }
        else if( S_ISREG( pointer->st_mode )) {
            printf( "%s is a regular file\n", fullpath );
        }
        else {
            printf( "%s is unknown!\n", fullpath );
        }
    }
    closedir(dirp);
}
```

File and Directory Permissions

Recall that:

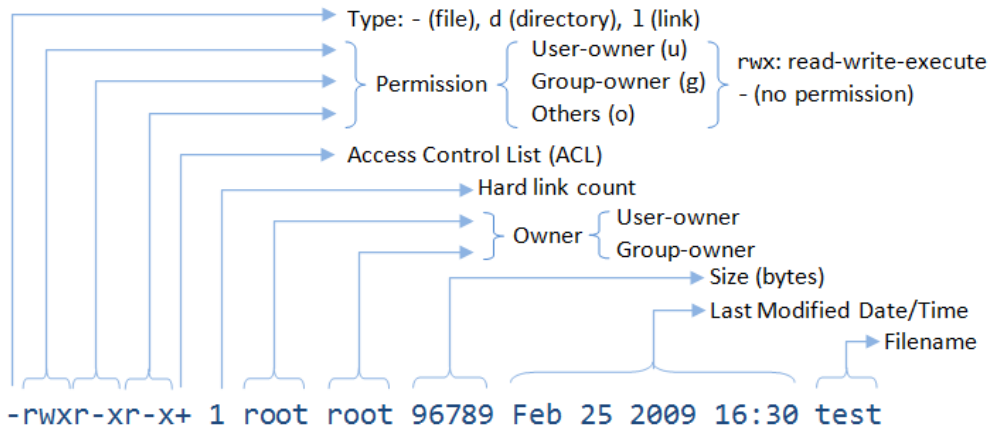
- Like most modern systems, Linux arranges its file system in a hierarchical structure of files and directories. Directories may contain entries for files and other directories.
- File entries contain the file's name, together with a pointer to a structure termed the *inode* (the *i*nformation node?), which represents the details of the file.

These are the familiar items shown by the standard *ls -l* command:

```
drwxr-xr-x   11 chris  staff   1024 Jul 29 20:29 WWW
-rw-------    1 chris  chris  53436 Aug  1 16:28 autonomous.pdf
-rw-r--r--    1 chris  chris     88 Dec 20  2016 scrolldown.gif
```

Multiple file entries, from possibly different directories, may point to the same inode. Thus, it is possible to have a single file with multiple names - we say that the names are *links* to the same file.

One unsigned 32-bit integer field in each inode contains the file's *permission (or protection) mode bits* - see */usr/include/bits/typesizes.h*



From history, the permission mode bits appear in the same integer defining the file's type (regular, directory, block device, socket, ...) -
See *man 2 stat* for details.

File and Directory Permissions, *continued*

When access requests are made by a *process* on behalf of a *subject* (a user) for an *object* (a file), the Unix kernel compares the *effective* user- and group-id attributes of the process against the permission mode bits of the file.

Of note, if the owner's permission bits of a file or directory are not set, then the owner cannot access the object by virtue of the 'group' or 'other' bits (can you think why?).

The inode structure also contains indication of the object's *setuid* and *setgid* status, together with a *sticky bit* having an overloaded meaning (historically, setting the sticky bit on an executable file requested that it not be swapped out of memory - requiring privilege to set the bit).

On different variants of Unix/Linux the permission mode bits, in combination, have some obscure meanings:

- having execute access, but not read access, to a directory still permits an attacker to 'guess' filenames therein,
- having the sticky bit set on a directory permits only the owner of a file, therein, to remove or modify the file,
- having the setgid bit set on a directory means that files created in the directory receive the groupid of the directory, and not of their creator (owner).

A system administrator managing different operating systems (Unix/Linux, macOS, many flavours of Windows) needs be aware of these subtle differences.

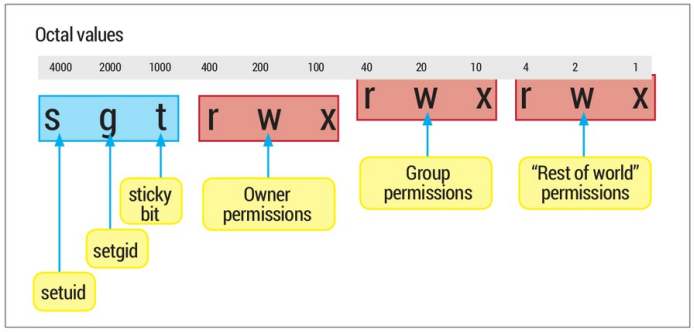
Describing permissions using *octal* values

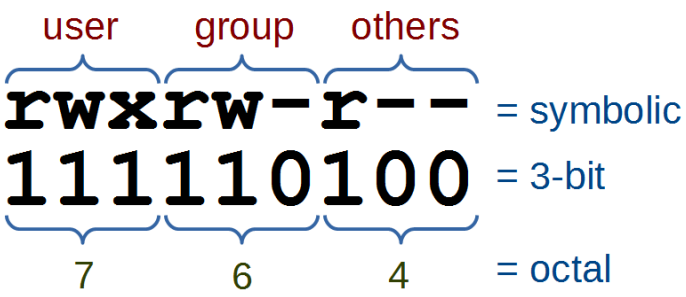
While we know that all data within a computer is stored in *binary*, we do not refer to or describe system-focused values with ones-and-zeroes.

In the case of file-system permissions, 3 bits (ranging over the integer values 0..7) are used to store Boolean values for the *read*, *write*, and *execute* permissions.

Further, 3 sets of these 3 permissions bits, one for each of the *user*, *group*, and *'other'* owners are employed for each file-system entry.

So, while we don't describe these data values in *binary*, it makes a lot of sense to describe them using*octal* values:





You may also like to try the Unix Permissions Calculator.

Accessing file-system permission bits

The previous examples, using RGB colour values, only employed individual *bytes* of the integers being considered.

Here we need to access *individual bits*, using each bit as if it were a Boolean value (which C11 doesn't directly support).

Recall that we have seen how we can determine if a directory entry is another directory or a file:

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <unistd.h>

  .....
  struct stat  stat_buffer;

  if(stat(pathname, &stat_buffer) != 0) {
     perror( progname );
  }
  else if( S_ISDIR( stat_buffer.st_mode )) {
    printf( "%s is a directory\n", pathname );
  }
  else if( S_ISREG( stat_buffer.st_mode )) {
    printf( "%s is a file\n", pathname );
  }
  else {
    printf( "%s is unknown!\n", pathname );
  }
```

```
// DEFINE A "NEW" TYPE TO REPRESENT A FILE'S MODE
typedef unsigned int mode_t;

struct stat {
    ....
    mode_t  st_mode;
    ....
}

#define S_IFMT        0170000 // BITS DETERMINING FILE TYPE

// File types
#define S_IFDIR       0040000 // DIRECTORY
....
#define S_IFREG       0100000 // REGULAR FILE

// MACROS FOR TESTING FOR DIFFERENT FILE TYPES
#define S_ISDIR(mode)    (((mode) & S_IFMT) == (S_IFDIR))
....
#define S_ISREG(mode)    (((mode) & S_IFMT) == (S_IFREG))
```

This C program better explains demonstrates these concepts: showmode.c