CS107 Guide to C stdlib functions

Guide to C stdlib functions

Prepared by Steve Choi, revised by Pablo Ceballos, CS107 TAs

The term "standard library" refers to the collection of functions that are packaged with a programming language. The functionality provided can be extensive (such as Java's packages for graphics/networking/database or C++ STL containers and algorithms) but you'll find C's standard library is quite minimal. It consists of dozen or so interfaces (header files) which provide facilities for reading/writing files, simple string/char handling, sort/search, generating random numbers, and primitive features for handling exceptional conditions. All of the modules are grouped into one library known as `libc`; this library is linked by default when building an executable. There is a separate library `libm` that contains implementation of the math operations declared in `math.h`. `libm` must be explicitly linked to resolve its symbols.

Here is our overview of the functions most relevant to CS107, grouped by header file. Our list is not exhaustive and skimps on details, so plan to refer to man pages and/or a C reference book where you need additional information.

<stdio.h> standard input/output

Reading and writing information in C is done using file pointers, which are variables of type `FILE*`. You can open a file by name to get a file pointer or use one of the three pre-opened global file pointers: `stdin` (console input), `stdout` (console output), or `stderr` (console error).

Printing, output

The reading and writing functions come in three different varieties based on how you are processing: character-by-character, line-by-line, or formatted. I listed the formatted functions first, as they are the workhorses you will use for most purposes, but the character and line functions are also given here.

```
int fprintf(FILE *fp, const char *format, ...)
int printf(const char *format, ...)
```

`fprintf` writes formatted text to the given file pointer (`printf` writes to stdout). The format string can contain a combination of literal characters (which are output unchanged) and format specifiers (e.g. %c for characters, %d for decimal values, %s for strings, and so on), which are placeholders in the output. The additional arguments are the values to substitute, one value for each format specifier in the order of occurrence in the format string. Note that printf is an example of a *variable-argument* function. The number of arguments is not constant, it depends on how many format specifiers are

being used. The format specifiers have various options that allow fine-tuning the format (field width, precision, alignment, etc). Don't worry about memorizing these details, plan to look them up on as-needed basis. The return value is the number of characters written.

```
int putchar(int ch)
int putc(int ch, FILE *fp)
```

`putchar` and `putc` write a single character either to the specified file pointer or stdout for putchar. The return value reports success/error.

```
int fputs(const char *s, FILE *fp)
int puts(const char *)
```

`fputs` writes a line of text to the given file pointer (`puts` writes to stdout). The return value reports success/error.

Reading, input

```
int fscanf(FILE *fp, const char *format, ...)
int scanf(const char *format, ...)
```

`fscanf` reads formatted text from the given file pointer (`scanf` reads from `stdin`). The format string can contain a combination of literal characters and conversion specifiers (%), which are used as placeholders to read values from the input. A formatted read pulls characters from the file stream and attempts to match them to the format string, filling in placeholders as it goes. It will stop at the first mismatch, otherwise it will match to the end of the format string. Understanding how the format string is matched to the input can be tricky. Here are the basic rules:

- any amount of whitespace within the format string will match any amount of whitespace in the input
- literal characters in the format string have to match the input exactly (any mismatch halts scanning)
- each conversion specifier must match a correctly formatted value in the input (what is "correct" depends on specifier, i.e %d matches a non-empty sequence of digits)

The additional arguments past the format string are the variables to read into, one for each conversion specifier in the order of occurrence in the format string. These arguments **must be passed by reference,** i.e. if the conversion specifier is %d, the matching argument must be a **pointer to an integer**. The return value is the number of successful conversions made and this value should be checked to verify the success of the scanning. There is a lot of complexity buried within scanf. Just as with the specifiers to printf, there are options that fine-tune the format (field width, delimiters, etc.) that you can investigate on an as-needed basis.

```
int getchar()
int fgetc(FILE *fp)
```

`fgetc` reads the next character from the given file pointer (`getchar` reads from stdin). The return value is the character read or EOF if no characters remain.

```
char *fgets(char buf[], int buflen, FILE *fp)
```

`fgets` reads the next line of text from the given file pointer. Characters are read from the file, stopping at the first newline or EOF or after buflen-1 characters. The characters are written to buf, which is expected to have sufficient memory to store buflen characters (including null terminator). The return value is the address of first char in buf on success, NULL is returned on EOF or error.

```
int feof(FILE *fp);
```

`feof` reports whether the EOF condition has been set for `fp` . The EOF condition is triggered when attempting to read from a file when no more characters remain to be read. Note that `feof` does not report whether a subsequent read *would* fail due to EOF, it only reports that a previous read *did* fail/stop due to EOF. You cannot reliably check for EOF before reading; instead you attempt to read and then check afterwards.

File operations

```
FILE *fopen(const char *filename, const char *mode)
```

`fopen` opens a file with the given filename and mode (usually "w" for write or "r" for read -- see man page for other modes). Returns a file pointer or NULL on failure.

```
int fflush(FILE *fp)
```

`fflush` flushes any buffered writes on a file pointer out to console/disk. The return value is 0 (success) or EOF (faliure).

```
int fclose(FILE *fp)
```

`fclose` is used to close a file pointer and free any memory.

String input/output

```
int sprintf(char *s, const char *format, ...)
int sscanf(const char *s, const char *format, ...)
```

The sprintf/sscanf functions perform the same formatted printf/scanf operations as above, but read the input from or write the output to a string, instead of a file.

<stdlib.h> standard library utilities

Utility functions

```
int rand()
void srand(unsigned int seed)
```

This psuedo-random number generator `rand()` returns a random value between 0 and RAND_MAX. Use `srand()` function to seed the generator if needed.

```
void qsort(void *base, size_t nmemb, size_t elemsz, int (*compar)(const void *,
const void *))
```

This is a generic implementation of quicksort that uses a `void*` interface. Note the last argument is function pointer, which is where the client supplies their comparator callback function that uses a `void*` interface.

The standard comparison callback used by `qsort/bsearch/lfind` takes two `const void *` parameters (pointers to two elements) and is expected to return a negative, zero, or positive return value to indicate whether the first is less than than the second (negative), the first is equal to the second (zero), or the first is greater than the second (positive). A callback should be symmetric, i.e., both arguments are same type, and result from cmp(a, b) is the inverse of cmp(b, a).

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t elemsz, int
(*compar)(const void *, const void *))
```

Performs binary search on a generic array using a `void*` interface. The last argument is a function pointer to a standard comparator callback function. The elements in the array must already be ordered in ascending order according to the client's comparator function. The return value is a pointer to the matched element or NULL if not found.

From non-standard GNU header <search.h>

```
void   *lfind(const void *key, const void *base, size_t *nmemb, size_t elemsz,
int (*compar)(const void *, const void *))
void *lsearch(const void *key, void *base, size_t *nmemb, size_t elemsz, int
(*compar)(const void *, const void *))
```

Perform linear search on a generic array using a `void*` interface. The last argument is a function pointer to a standard comparator callback function. The return value is a pointer to the matched element or NULL if not found. The third parameter (num elements) is passed by reference to `lsearch` because it will append the element to the array if not found and increment the number of elements. The third parameter (num elements) is similarly passed by reference to `lfind` despite the fact that it doesn't change the count. I assume this decision was motivated by the desire to maintain a parallel interface between the two routines, but it seems pretty bogus to me. At the very least, it should have qualified the pointee as `const`, sigh.... Consider it just an annoying quirk of `lfind`.

`lfind` and `lsearch` are GNU extensions. The GNU libc contains the ANSI standard routines, plus many others. lfind/lsearch/strdup/etc. are not present in standard C but can be nice conveniences and ubiquitous enough that we encourage their use despite being introducing mild portability issues due to being non-standard.

Dynamic memory management

```
void *malloc(size_t size)
```

`malloc` allocates a new block of heap memory. Returns a pointer to at least size bytes of memory on the heap or NULL on error.

```
void *calloc(size_t num, size_t size)
```

`calloc' allocates a new block of heap memory and initialize contents to zero. Returns a pointer to at num*size bytes of zero-filled memory on the heap. Returns NULL on error.

```
void *realloc(void *p, size_t size)
```

`realloc` is used to resize a block of heap memory. It is given a previously allocated pointer to heap memory and a new size and returns a pointer to heap memory that is at least size bytes and contains the same contents as the original heap block. The returned pointer may or may not be the same as the original. Returns NULL on error.

```
void free(void *p)
```

Free a block of heap memory.

Program control

```
void exit(int status)
```

Halts program without any further execution and returns status as exit code. Typically 0 is used to indicate success and small positive numbers for various error outcomes. The non-standard GNU extension `error` combines printing a message with exit, see its man page for information.

<string.h> string functions

A string in C (sometimes just called a C-string) is merely a pointer to a sequence of characters terminated with a null char. Given the many pitfalls associated with pointers, it is not surprising that strings can be one of the more treacherous parts of C. It is the client's responsibility to properly allocate the memory for strings and take care to ensure a string null-terminated. The string functions do not raise helpful errors on misuse such as accessing out of range, missing terminators, underallocated memory, and so on. If used improperly, the functions will be blunder through the request, leading to data corruption and/or crashes. Programmer beware!

```
size_t strlen(const char *s)
```

`strlen` returns the length of a string (not counting the null terminator).

```
char *strcpy(char *dest, const char *src)
```

`strcpy` copies the characters pointed to by src into the memory pointed to by dest. src must point to a valid, null-terminated sequence of characters and dest should point to valid memory of at least strlen(src) + 1 bytes. The return value is value of dest (so not

that useful).

```
char *strncpy(char *dest, const char *src, size_t n)
```

This variant of strcpy copies only the first n characters from src; therefore, dest only needs to point to memory of at least n + 1 bytes. If n > strlen(src), strncpy is smart enough to not run off the end of the src array. If there is a null terminator in the first n characters of src, it will be copied to dest, otherwise strncpy does **not** null-terminate dest.

```
char *strcat(char *dest, const char *src)
```

`strcat` appends src string to the end of the dest string overwriting the null character at the end of the dest string and adding a new null character at the end of the newly appended string. Both src and dest should be properly null-terminated strings and there should be sufficient memory at the end of dest to accommodate the additional characters from src. The return value is value of dest (so not that useful).

```
char *strncat(char *dest, const char *src, size_t n)
```

This variant of strcat appends only the first n characters from src. strncat always null terminates dest (note difference compared to strncpy).

```
int strcmp(const char *s1, const char *s2)
```

`strcmp` compares two strings lexicographically. Returns 0 if they are same, a negative value if s1 < s2, and a positive value otherwise.

```
int strncmp(const char *s1, const char *s2, size_t n)
```

This variant of strcmp only compares the first n characters of each string (can be used to compare prefix/substring).

The GNU libc also includes convenient case-insensitive variants `strcasecmp` and `strncasecmp` (not standard C).

```
char *strchr(const char *s, int c)
char *strrchr(const char *s, int c)
```

These two functions search a string for a given character and returns a pointer to first occurrence found, or NULL if not found. `strchr` searches left to right, `strrchr` from right to left.

```
char *strstr(const char *haystack, const char *needle)
```

`strstr` searches for the first occurrence of the substring `needle` in the string `haystack`. Returns a pointer to the occurrence found within `haystack` or NULL if no occurrence found.

```
char *strdup(const char *s)
```

The `strdup` function is not standard C, but this minor convenience function is a common addition offered by the GNU libc. It returns a new heap-allocated copy of the given string (equivalent to malloc + strcpy). It is an oddball in that it allocates heap memory "secretly", forcing client to know to free without having explicitly malloc'ed themselves.

`strspn` answers the question, "How many places can we go in the first string `s` before I encounter a character not in the second string `accept`?". It returns a count. The ordering of characters in the second string does not matter:

```
size_t strspn(const char *s, const char *accept);
```

`strcspn` is the opposite - it answer the question, "How many places can we go in the first string `s` before I encounter a character in the second string `reject`?" It returns a count. The ordering of characters in the second string does not matter.

```
size_t strcspn(const char *s, const char *reject);
```

Data operations

```
void *memcpy(void *dest, const void *src, size_t n)
void *memmove(void *dest, const void *src, size_t n)
```

These functions copy raw data from one memory location to another. Both copy n bytes from the memory pointed to by src to the memory pointed to by dest. The pointers src and dest should refer to valid memory at least n bytes long. The memmove variant should be used when the src and dest regions overlap, memcpy is not guaranteed to work correctly in such a case. Both functions return the dest pointer.

```
void *memset(void *dest, int ch, size_t n)
```

`memset` repeatedly writes the given character ch into each of the first n bytes of memory pointed to by dest. Returns dest pointer.

<ctype.h> char functions

For largely historical reasons, the ctype functions use the type `int` for the arguments and return values, even though those values are actually treated as `char`.

```
int isdigit(int ch)
int isalpha(int ch)
int isupper(int ch)... and many others (see man page for ctype)
```

The ctype `isXXX` functions classify a given character `ch` as alphabetic, punctuation, etc. The return value is non-zero for true, 0 for false.

```
int toupper(int ch)
int tolower(int ch)
```

These functions return the upper/lower case equivalent of character `ch` (or return `ch` unchanged if not a alphabetic letter).

<assert.h> assertions

```
assert(expr)      / /btw, assert is a preprocessor macro, not a function
```

Verifies the given expression evaluates to true. If not, halts the program and prints a diagnostic message about the failed assertion. It can be helpful to use asserts during development to verify required conditions. Should an exceptional situation arise, the failed assert will immediately draw your attention to the problematic condition. Note that assert is a diagnostic communication between programmers (or programmer-to-self). They are not appropriate for communicating to the user -- when you are informing the user how to properly use the program, better to use your own error-handling scheme with helpfully-worded print statements.
