The C Standard Library

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Language Semantics and Extensibility

- The "C language" specifies what statements in the language mean – it's *syntax* and *semantics*.
- A C compiler translates statements in the C language into assembler/machine code whose effect is the same as the meaning of the C statements
- Functions/subroutines are an "extensibility mechanism"
  - They're a way to introduce new statement meanings
- As in other languages, a C program uses a combination of language features and functions
  - Functions are often distributed as libraries

C Standard Library

- From Wikipedia: https://en.wikipedia.org/wiki/C_standard_library

- The **C standard library** or **libc** is the standard library for the C programming language, as specified in the ISO C standard.[1] Starting from the original ANSI C standard, it was developed at the same time as the C library POSIX specification, which is a superset of it.[2][3] Since ANSI C was adopted by the International Organization for Standardization,[4] the C standard library is also called the **ISO C library**.

What Did That Mean

- There are many C compilers, so…
- Define a standard set of useful functions that all of them should/must implement
  - That way, if you implement your app using only C language and standard library functions, your code is "portable" (by re-compiling/re-linking)
- But…
- There's more than one standard, and…
- It's common for individual compilers/systems to provide supersets of the standard
  - When a program uses one of the extended functions, it's no longer portable
- What standard a standard library function is standard in doesn't leap up off the man page

man strchr

- STRCHR(3) Linux Programmer's Manual STRCHR(3)

- NAME
- strchr, strrchr, strchrnul - locate character in string

- SYNOPSIS
- #include <string.h>
- char *strchr(const char *s, int c);
- char *strrchr(const char *s, int c);
- #define _GNU_SOURCE /* See feature_test_macros(7) */
- #include <string.h>
- char *strchrnul(const char *s, int c);
- …

- CONFORMING TO
- strchr(), strrchr(): POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD.
- strchrnul() is a GNU extension.

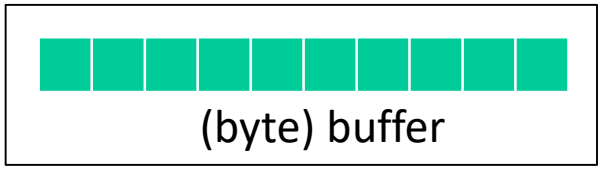
Function vs. Macro vs. Compiler Internal

- A standard function can be a normal function, written in C (or assembler)
  - Invoking it requires a procedure call (which is expensive)
- A standard function can be implemented as a pre-processor macro
  - Invoking the function results in code being inserted into your program
  - No procedure call overhead
- The compiler can internalize standard functions and emit code for them
  - Similar in intent to the pre-processor macro approach

#include <stdio.h>

- printf / fprintf / sprintf
  - Print to stdout, a particular file, or a string (array of char)
  - It's printing, it's also conversion to string
  - Returns number of bytes printed
- scanf / fscanf / sscanf
  - Read from stdin, a particular file, or the contents of an array of char
  - It's conversion from "string" to other data types
  - Returns number of items read
- fopen / fclose (vs. open / close)
  - Open/close file for "buffered reading"

#include <stdio.h>



- fread / fwrite
- fseek / fgetpos
- feof / ferror
- fflush
- getchar / getc / fgetc / putchar / putc / fputc
- Ungetc
- gets / fgets / puts / fputs

- perror (for system error returns)

#include <string.h>

- strlen
- strcpy / strncpy / strcat / strncat
- strcmp (returns -1, 0, or 1 for less than, equal, greater than)
- strchr / strstr / strrchr / strrstr
- strtok
- strerror (for system errors)

- memcpy / memove
- memcmp / memmem
- memset

#include <stdlib.h>

- atoi / atol / atof
- strtol / strtoul / strtod (like above, but better able to indicate errors)
- malloc / calloc / realloc / free
- getenv / setenv
- system (causes execution of a command as thought it were given to a shell)
- qsort / bsearch

#include <assert.h>

- assert
  - int getX(MyType *pObj) {
    assert(pObj != NULL);
    …
- assert() is a pre-processor macro
- If symbol NDEBUG is NOT defined, preprocessor produces code that tests condition and complains/exits if it's false
- If symbol NDEBUG is defined, the preprocessor emits nothing
  - So, no run time overhead (e.g., to test the condition)
- Let's you leave debugging code in your code without performance penalty
- Warning: do not use assert for essential sanity checks

#include <limits.h>

- CHAR_MIN / CHAR_MAX / UCHAR_MAX
- INT_MIN / INT_MAX / UINT_MAX
- LONG_MIN / LONG_MAX / ULONG_MAX
- Etc.

#include <math.h>

- acos / asin / atan / cos / cosh / sin /sinh/ tanh
- exp / log / log10
- pow / sqrt / floor / fmod
- …

#include <ctype.h>

- isupper / islower / toupper / tolower
- isalpha / isdigit / isalnum / ispunct / isspace / ispunct

#include <time.h>

- Current time
- Elapsed time

#include <errno.h>

C doesn't have exceptions

- extern int errno;

- Many routines that can fail in different ways don't/can't indicate just how they failed
  - Perhaps they just return NULL to indicate unhappy, but that's only one value for everything that can go wrong
  - *"The fopen(), fdopen() and freopen() functions may also fail and set errno for any of the errors specified for the routine malloc(3)."*

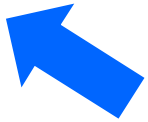
- Many of these routines set this global (errno) to an error specific value
- perror() will print a string representation of the error
- strerror() will provide a string representation of the error

Additional errno details

- From the man page:
  - *The value in errno is significant only when the return value of the call indicated an error (i.e., -1 from most system calls; -1 or NULL from most library functions); a function that succeeds is allowed to change errno. The value of errno is never set to zero by any system call or library function.*
  - *errno is defined by the ISO C standard to be a modifiable lvalue of type int, and must not be explicitly declared; errno may be a macro. errno is thread-local; setting it in one thread does not affect its value in any other thread.*

#include <stdarg.h>

- For "variadic" methods like
  int printf(const char*, …);



- va_start , va_arg, va_end

```
va_list ap;
va_start(ap, last_named_parameter);
while (i = va_arg(ap, int) ) { … }
va_end(ap);
```

- This is a pretty terrible idea…
  - "If there is no next argument, or if type is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur."
  - "If *ap* is passed to a function that uses **va_arg(***ap*,*type***)** then the value of *ap* is undefined after the return of that function."
- Although at the same time it seems natural enough. Cf. printf().

Aside: Method Polymorphism

- Polymorphism: the ability to associate a single name with many different types (here, of functions)
  - Java Example: void update(int); *and* void update(String);
  - Example: any base class method overridden by derived class

- C does not have method polymorphism
  - All method names are in the global namespace
  - There can be only one defined object with a particular name in a single namespace

- C approximations
  - function pointers, if all instances have the same arg list and return type
  - va_start, va_arg, va_end

Static vs. Dynamic Polymorphism

- "Static": at compile time
  - no runtime overhead
  - update(6) / update("new address")
    - Compiler can "see" which version you want by the argument types
- "Dynamic": at run time
  - Not always possible to determine at compile time what the argument types will be
- In Java, dynamic dispatch is implemented by the compiler / JVM
  - Reliable
- In C, you're given some hammers and nails and allowed to build whatever you want

#include <unistd.h>

- "The <unistd.h> header defines miscellaneous symbolic constants and types, and declares miscellaneous functions."
  - POSIX OS API
- ```
  int          access(const char *, int);
  unsigned     alarm(unsigned);
  int          chdir(const char *);
  int          chown(const char *, uid_t, gid_t);
  int          close(int);
  size_t       confstr(int, char *, size_t);
  char         *crypt(const char *, const char *);
  int          dup(int);
  int          dup2(int, int);
  void         _exit(int);
  void         encrypt(char [64], int);
  int          execl(const char *, const char *, ...);
  Etc.
  ```
  ```

#include <unistd.h>

- getopt and getoptlong
  - For processing command line arguments
- Crude example:

```
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 1;
      default: abort ();
}
```

https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html

#include <unistd.h>

- Provides a second interface to streams (files)
  - The other was <stdio.h> which provides FILE*
- This is the "hammer and nails" interface

- int open(const char *pathname, int flags, mode_t mode);
  ssize_t read(int fd, void *buf, size_t count);
  ssize_t write(int fd, const void *buf, size_t count);
  int close(int fd);
  ...