C Standard Library

- Overview
- Standard C99 Libraries
- Standard C99 Libraries Continued
- Standard C99 Libraries Continued
- Assertions
- Assertions Log
- Non-Local Goto’s
- Balanced Parentheses Checking
- Balanced Parentheses Checking Continued
- Balanced Parentheses Checking Continued
- Balanced Parentheses Checking Continued
- Balanced Parentheses Log
- Strings and Character Types
- Strings and Character Types Palindrome Example
- Palindrome Example Continued
- Palindrome Log
- Standard I/O Introduction
- Opening a Stream
- Opening a Stream Continued
- Truncating Files Example
- Buffering
- Buffering Control
- Buffering Example
- Buffering Example Log
- Stream I/O
- Character I/O
- `wc(1)` Program
- `wc(1)` Program Continued
- `wc(1)` `main()`
- `wc(1)` Log
- Line I/O
- Binary I/O
- Formatted Input
- Formatted Output
- Error Detection
- Standard I/O Efficiency
- Stream Positioning
- Non-Character I/O Example
- Non-Character I/O Example Continued
- Non-Character I/O Example Log
- Updating a File

- Temporary Files
- Temporary Files Continued
- Temporary Files Example
- Temporary Files Log
- References

Overview

- Review all standard C99 libraries.
- The use of `assert()` with an example.
- The use of `setjmp()` and `longjmp()` with an example.
- Use of strings and character types routines.
- Remainder concentrates on standard I/O.

Standard C99 Libraries

`assert.h`
: Conditional `assert`.

`complex.h`
: Complex number math. C99.

`ctype.h`
: Character classification/conversion macros. `isXXXXX()`, `toupper()`, `tolower()`.

`errno.h`
: `errno` global variable or macro; Error-number constants `E*` like `EDOM` and `ERANGE`.

`fenv.h`
: Control floating point environment. C99.

`float.h`
: `float`, `double` constants.

`inttypes.h`
: Formatting for min-sized ints. C99.

`iso646.h`
: Definition of macros for bitwise and logical operators which can be difficult to type on some keyboards. C89 Amendment 1.

Standard C99 Libraries Continued

`limits.h`
: Limits of integral types.

`locale.h`
: Facilitates L10N.

`math.h`
: Standard math functions.

`setjmp.h`
: Nonlocal jumps.

`signal.h`
: Signal handling.

`stdarg.h`
: Facilitates user-defined functions which take a variable number of arguments.

`stdbool.h`
: Defines boolean constants. C99.

`stddef.h`
: Some basic definitions including `NULL`, `offsetof`, `size_t`, etc.

Standard C99 Libraries Continued

`stdint.h`
: Portably define integers of exact or minimum guaranteed size. C99.

`stdio.h`
: Buffered I/O.

`stdlib.h`
: Utility routines, including memory allocation (`malloc()` and friends), `exit()`, string-to-number conversion routines.

`string.h`
: String manipulation.

`tgmath.h`
: Generic (with type parameter) math functions. C99.

`time.h`
: Time manipulation.

`wchar.h`
: Support for wide characters. C89, Amendment 1.

`wctype.h`
: Classification and conversion of wide characters. C89, Amendment 1.

Assertions

`assert(`*test*`)` aborts program with error if *test* is false. All assertions can be deactivated by defining preprocessor symbol `NDEBUG` during compilation.

Following program computes average of command-line integer arguments in ./programs/average.c:

```
static int
average(const char *ints[], int nInts)
{
  assert(nInts > 0);
  int sum = 0;
  for (int i = 0; i < nInts; i++) {
    sum += atoi(ints[i]);
  }
  return sum/nInts;
}

int
main(int argc, const char *argv[])
{
  printf("%d\n", average(&argv[1], argc - 1));
  return 0;
}
```

Assertions Log

```
$ gcc -g -Wall --std=c99 average.c -o average
$ ./average 5 -5 10
3
$ ./average
average: average.c:8: average: Assertion ‘nInts > 0’ failed.
Aborted
$ gcc -g -Wall --std=c99 -DNDEBUG average.c -o average
$ ./average
Floating point exception
$
```

Non-Local Goto's

Standard library facility in `setjmp.h` (see text, section 6.8):

- Calling `int setjmp(jmp_buf env)` remembers the current control context in `env` and returns 0.
- Subsequently, calling `longjmp(env, val)` causes the original call to `setjmp()` to return **again** with return value `val` (1 if `val` was 0).
- `setjmp()` must be used as test or within a test comparison.
- Undefined behavior if `longjmp()` is called after the function which called `setjmp()` has returned.
- Non-`volatile` local variables within the function calling `setjmp()` may have the wrong value after the `longjmp()` if optimized by the compiler. Use `volatile` if the value is important.

Balanced Parentheses Checking

In ./programs/balanced.c:

```
typedef struct {
  const char *string;
  int index;
  jmp_buf error_env;
} TestContext;

static TestContext
make_context(const char *string)
{
  TestContext context = { .string = string, .index = 0 };
  return context;
}
```

Balanced Parentheses Checking Continued

```
static inline char
peek(const TestContext *context)
{
  return context->string[context->index];
}

static void match(TestContext *context, char c) {
  if (peek(context) == c) {
    context->index++;
  }
  else {
    longjmp(context->error_env, 1);
  }
}
```

Balanced Parentheses Checking Continued

```
/** Return normally if string pointed to by contextP consists of
 *  balanced paren. */
static void
balanced(TestContext *contextP)
{
  if (peek(contextP) == '(') {
    match(contextP, '(');
    balanced(contextP);
    match(contextP, ')');
  }
}

/** Return normally if string pointed to by contextP is balanced paren. */
static void
balanced_string(TestContext *contextP)
{
  balanced(contextP);
  match(contextP, '\0');
}
```

Balanced Parentheses Checking Continued

```
int
main(int argc, const char **argv) {
  for (int i = 1; i < argc; i++) {
    const char *string = argv[i];
    TestContext context = make_context(string);
    if (setjmp(context.error_env) == 0) {
      balanced_string(&context);
    }
    else {
      fprintf(stderr, "%s unbalanced at index %d\n",
              string, context.index);
    }
  }
  return 0;
}
```

Balanced Parentheses Log

```
$ ./balanced '(())'
$ ./balanced '(()))'
(()) unbalanced at index 4
$ ./balanced '(()'
(() unbalanced at index 3
$ ./balanced '((a)'
((a) unbalanced at index 2
$ ./balanced ''
$
```

Strings and Character Types

- `string.h` provides support for `NUL`-terminated strings. Routines include string utilities, memory copy and comparison routines, string scanning routines.
- Note that code like `char *s_copy = malloc(strlen(s)); strcpy(s_copy, s);` is wrong as there is no space for the terminating `NUL` in `s_copy`. Use `malloc(strlen(s) + 1)`.
- `ctype.h` provides routines (often implemented as macros) for checking the type of a character like `isspace()`, `isalpha()`, `isalnum()`, `isctrl()` as well as character conversion routines `toupper()`, `tolower()`.

Strings and Character Types Palindrome Example

In ./programs/palindrome.c:

```
static void
normalize_string(const char *s, char *normalized)
{
  int n = strlen(s);
  int normalizedIndex = 0;
  for (int i = 0; i < n; i++) {
    if (isalnum(s[i])) {
      normalized[normalizedIndex++] = tolower(s[i]);
    }
  }
  normalized[normalizedIndex] = '\0';
}

static void
reverse_string(const char s[], char reverse[])
{
  int n = strlen(s);
  for (int i = 0; i < n; i++) {
    reverse[n - i - 1] = s[i];
  }
  reverse[n] = '\0';
}
```

Palindrome Example Continued

```
static bool
is_palindrome(const char *s)
{
  char normalized[strlen(s) + 1];
  normalize_string(s, normalized);
  char reversed[strlen(normalized) + 1];
  reverse_string(normalized, reversed);
  return (strcmp(normalized, reversed) == 0);
}

int
main(int argc, const char *argv[])
{
  for (int i = 1; i < argc; i++) {
    if (!is_palindrome(argv[i])) {
      fprintf(stderr, "arg %d ’%s’ is not a palindrome\n", i, argv[i]);
    }
  }
  return 0;
}
```

Palindrome Log

```
$ ./palindrome ’Able was I ere I saw Elba’ ’A man, a plan, a canal, Panama!’
$ ./palindrome ’Able was I ere I saw Elba1’ ’A man, a plan, a canala, Panama!’
arg 1 ’Able was I ere I saw Elba1’ is not a palindrome
arg 2 ’A man, a plan, a canala, Panama!’ is not a palindrome
$
```

Standard I/O Introduction

- Available on a wide variety of OS's.
- Handles buffering.
- Opening a file results in a `FILE` object.
- Built on top of I/O facilities of native OS.
- `FILE` object wraps native I/O facility, containing buffer pointer, buffer size, buffer count, error flag, etc.
- Standard I/O *streams*.
- Three standard streams `stdin`, `stdout` and `stderr` defined in `stdio.h` header file.

Opening a Stream

```
FILE *fopen(const char *pathname, const char *type);
```

- Returns `NULL` on error.
- `pathname` gives absolute or relative path of file to be opened.
- `type` specifies whether file should be opened for read, write or update, and whether file is text (newline translation) or binary (no newline translation).

  `r`
  : Open for reading.

  `w`
  : Truncate to 0 length, or create for writing.

  `a`
  : Append; open for writing at EOF or create for writing.

Opening a Stream Continued

- `r`, `w` or `a` `type` specification can be followed by `+` and/or `b` characters (in either order) to indicate *read and write* (*update*) and *binary* respectively.
- If file is opened for update, then buffers must be cleared before changing from read to write or vice-versa. Specifically,
  - To write after read `fseek`, `fsetpos` or `rewind` or hit EOF.
  - To read after write, `fflush`, `fseek`, `fsetpos` or `rewind`.
- `FILE *freopen(const char *pathname, const char *type, FILE *fp)` opens specified file `pathname` on specified stream `fp`, closing `fp` first if it is already open. Typically used to open a specified file as `stdin`, `stdout` or `stderr`.

Truncating Files Example

The following program truncates a file:

```
int
main(int argc, const char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");
    if (!f || fclose(f) != 0) {
      fprintf(stderr, "could not read %s: %s\n",
              argv[i], strerror(errno));
      continue;
    }
    f = fopen(argv[i], "w");
    if (!f || fclose(f) != 0) {
      fprintf(stderr, "could not truncate %s: %s\n",
              argv[i], strerror(errno));
    }
  }
  return 0;
}
```

Buffering

- **Fully Buffered**: I/O occurs only when buffer is empty or full. Disk files are normally fully buffered.
- **Line Buffered**: I/O performed whenever a newline character is encountered on input or output. Terminals are normally line buffered.
- **Unbuffered**: I/O occurs as characters are read or written.
- `stderr` is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; else they are fully buffered.
- Default buffering can be changed.

Buffering Control

`void setbuf(FILE *fp, char *buf);`

- `setbuf` turns on buffering (fully buffered or line buffered) using `buf` as buffer of size `BUFSIZ`.
- `setbuf` can be used to turn off buffering by specifying `buf` as `NULL`.

`void setvbuf(FILE *fp, char *buf, int mode, size_t size);`

- `mode` can be `_IOFBF` for fully buffered, `_IOLBF` for line buffered, `_IONBF` for unbuffered.
- `buf` and `size` ignored for `_IONBF`.
- For `_IOFBF` or `_IOLBF`, if `buf` is `NULL`, then a system buffer of an appropriate length will be allocated.

`int fflush(FILE *fp)` forces a write of all buffered data for stream `fp`.

Buffering Example

The following program demonstrates fully-buffered versus line-buffered depending on whether or not a command-line argument is specified.

```
int
main(int argc, char *argv[])
{
  int bufMode = (argc > 1) ? _IOFBF : _IOLBF;
  char line[80];
  setvbuf(stdout, NULL, bufMode, 0);
  while (fprintf(stderr, "xx> ") >= 0 &&
         fgets(line, sizeof(line), stdin)) {
    fputs(line, stdout);
  }
  return 0;
}
```

Buffering Example Log

```
$ ./buffer
xx> This is line 1
This is line 1
xx> This is line 2
This is line 2
xx> $ ./buffer 1
xx> This is line 1
xx> This is line 2
xx> This is line 1
This is line 2
$
```

Stream I/O

- Character at a time I/O: `fgetc()`, `fputc()`, etc.
- Line at a time I/O: `fgets()`, `fputs()`, etc.
- Direct binary I/O: `fread()`, `fwrite()`.
- Formatted I/O: `scanf()`, `printf()`, etc.

Character I/O

- `int fgetc(FILE *fp)` returns next character from stream `fp`; Returns `EOF` (-1) on EOF or error.
- `int getc(FILE *fp)` is similar to `fgetc()` but may be implemented as a macro. This may be faster than `fgetc()`. Watch out for side-effects in argument like `getc(files[i++])`.
- `int getchar(void)` is equivalent to `getc(stdin)`.
- `int ungetc(int c, FILE *fp)` puts character `c` as next character to be read from stream `fp`. Only upto 1 character of consecutive pushback is guaranteed.
- `int fputc(int c, FILE *fp)`, `int putc(int c, FILE *fp)` and `putchar(int c)` are corresponding output routines. Returns `c` or `EOF` on error.

wc(1) Program

The following program is a simple word-count (wc(1)) program:

```
enum { COUNT_CHARS= 7 };

static int nCTotal = 0;         /* total # of chars */
static int nWTotal = 0;         /* total # of words */
static int nLTotal = 0;         /* total # of lines */

static void
wc(const char *fName, FILE *f)
{
  int inWord = 0;
  int nC = 0, nW = 0, nL = 0;
  int c;
  while ((c = getc(f)) != EOF) {
    nC++;
    if (c == '\n') nL++;
    if (isspace(c)) {
      inWord = 0;
    }
    else {
      if (!inWord) nW++;
      inWord = 1;
    }
  }
```

wc(1) Program Continued

Continuation of wc routine:

```
  if (ferror(f)) {
    fprintf(stderr, "i/o error: %s\n", strerror(errno));
  }
  printf("%*d %*d %*d %s\n", COUNT_CHARS, nL,
         COUNT_CHARS, nW, COUNT_CHARS, nC, fName);
  nCTotal += nC; nWTotal += nW; nLTotal += nL;
}
```

wc(1) main()

`main()` routine for wc program:

```
int
main(int argc, const char *argv[])
{
  int i;
  if (argc == 1) {
    wc("", stdin);
  }
  else {
    for (i = 1; i < argc; i++) {
      FILE *in = fopen(argv[i], "r");
      if (!in) {
        fprintf(stderr, "could not read %s: %s\n",
                argv[i], strerror(errno));
      }
      else {
        wc(argv[i], in);
        fclose(in);
      }
    }
    if (argc > 2) {
      printf("%*d %*d %*d total\n", COUNT_CHARS, nLTotal,
             COUNT_CHARS, nWTotal, COUNT_CHARS, nCTotal);
    }
  }
  return 0;
}
```

wc(1) Log

```
$ which wc
/usr/bin/wc
$ wc wc.c
     62     183    1154 wc.c
$ ./wc wc.c
     62     183    1154 wc.c
$ wc wc.c buffer.c
     62     183    1154 wc.c
     14      41     278 buffer.c
     76     224    1432 total
$ wc <wc.c
     62     183    1154
$ ./wc <wc.c
     62     183    1154
$
```

Line I/O

- `char *fgets(char *buf, int n, FILE *fp)` reads upto next newline but no more than `n - 1` characters into `buf`, terminating buffer read with a `NUL '\0'`. Returns `buf` on success; `NULL` on error or EOF.
- `char *gets(char *buf)` reads next line into buffer, discards terminating `\n` and terminates with `NUL`. Deprecated and notorious.
- `int fputs(char *str, FILE *fp)` writes contents of `NUL`-terminated string `str` (which may or may not contain a `'\n'`) to stream `fp`. Returns non-negative number on success, `EOF` on error.
- `int puts(char *str)` writes `NUL`-terminated string `str` plus `'\n'` to `stdout`.

Binary I/O

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

- Used to read or write `nobj` objects of size `size` pointed to by `ptr` from or to stream `fp`.
- Returns number of objects read or written.
- Can be used to write arrays of `char`s or even arrays of general `struct`s.
- Portability problems in writing a file on one system and reading it back on another, because of non-portable structure alignments and data-type formats.

Formatted Input

- `int scanf(const char *fmt, ...)` required **address** of arguments.
- `scanf()` is a specialization of `fscanf()`.
- `sscanf()` reads from string instead of a file. Useful for converting data from string to binary.
- `fmt` specifiers bear superficial resemblance to `printf()` `fmt` specifiers.
- Returns number of arguments converted.

Formatted Output

- `int printf(const char *fmt, ...)`
- `printf()` is a specialization of `fprintf()`.
- `sprintf()` writes to a string instead of a file. Useful for converting data from binary to string.
- If format precision or width is specified as `*`, then precision or width is taken from an argument.
- `fmt` specifiers bear superficial resemblance to `scanf()` `fmt` specifiers.
- Returns number of characters output.

Error Detection

- Many routines return `EOF` on either real EOF or on error.
- `int feof(FILE *fp)` returns non-zero if stream `fp` is at EOF.
- `int ferror(FILE *fp)` returns non-zero if stream `fp` has encountered an error.
- `void clearerr(FILE *fp)` clears both EOF and error conditions for stream `fp`.

Standard I/O Efficiency

- Library takes care of all buffering.
- Function calls (entirely in user-space) are usually more efficient than system calls.
- Macro calls are often more efficient than function calls, but can lead to larger code.
- Best to let library choose buffer size.

Stream Positioning

- `long ftell(FILE *fp)` returns position in stream `fp`.
- `int fseek(FILE *fp, long offset, int whence)` sets position in stream `fp` to that specified by `offset` and `whence`, where the interpretation of `whence` (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`) is the same as for `lseek()`.
- `void rewind(FILE *fp)` sets position to the beginning of stream `fp`.
- For text files, they can be positioned at the start or at a value returned by `ftell()`.
- ANSI C introduced `int fgetpos(FILE *fp, fpos_t *pos)` and `int fsetpos(FILE *fp, const fpos_t *pos)`.

Non-Character I/O Example

The following program tracks the number of times it has been invoked using an auxiliary binary file:

```
int
main(int argc, const char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "%s COUNT_FILE_NAME\n", argv[0]);
    exit(1);
  }
  {
    int count = 0;
    FILE *f = fopen(argv[1], "rb+");
    if (f) {
      if (fread(&count, sizeof(int), 1, f) != 1) {
        fprintf(stderr, "i/o error: %s\n", strerror(errno));
        exit(1);
      }
    }
    else {
      f = fopen(argv[1], "w");
    }
    if (!f) {
      fprintf(stderr, "could not open %s: %s\n",
              argv[1], strerror(errno));
    }
```

Non-Character I/O Example Continued

```
    rewind(f);
    printf("%s: %d\n", argv[1], count);
    count++;
    if (fwrite(&count, sizeof(count), 1, f) != 1) {
      fprintf(stderr, "i/o error: %s\n", strerror(errno));
      exit(1);
    }
  }
  return 0;
}
```

Non-Character I/O Example Log

```
$ ./progcount progcount.dat
progcount.dat: 0
$ ./progcount progcount.dat
progcount.dat: 1
$ ./progcount progcount.dat
progcount.dat: 2
$ ./progcount progcount.dat
progcount.dat: 3
$ ./progcount progcount.dat
progcount.dat: 4
$ od progcount.dat
0000000 000005 000000
0000004
$
```

Updating a File

- Cannot stretch a file directly.
- Common idiom for updating a file:

  1. Read source file to get data for update.
  2. Write results of update to a temporary file (usually concurrent with (1)).
  3. Rename temporary file to source file name.

  An advantage of this idiom is that it minimizes the potential for data loss.

Temporary Files

Typically, the environmental variable `TMPDIR` controls the directory where the following calls create temporary files.

- `char *tmpnam(char *ptr)` returns a name different from that for any existing file. Stores it in a static area if `ptr` is `NULL`. Otherwise it stores it in the buffer pointed to by `ptr` (which must have space for at least `L_tmpnam` characters). Returns `NULL` on failure. Not thread-safe if ptr is NULL. No guarantee that some other program has not created a file with returned name by the time caller creates it. Another problem is that, since Unix has a limitation of being unable to rename files across arbitrary directories, this call cannot be used to create the temporary file used in the updating file idiom.

- `FILE *tmpfile(void)` creates and opens a new temporary file with type `w+b`. The file is deleted when closed or upon program termination.

Temporary Files Continued

- `int mkstemp(char *template)` returns a file descriptor for a temporary file with name formed from `template`. `template` must be provided with a suffix of 6 `X`'s. These `X`'s are replaced to form a unique temporary name and a new file with that name is opened for read/write atomically. Recommended method. Allows caller to control the directory where the temporary file is created (via `template`) and there is no race condition between creation of the name and the file.

Temporary Files Example

The following program illustrates the use of temporary files:

```
enum {MAXLINE = 120 };

int
main(void)
{
  char name[L_tmpnam], line[MAXLINE];
  FILE *fp;

  printf("%s\n", tmpnam(NULL)); /* first name */

  tmpnam(name);                 /* second name */
  printf("%s\n", name);

  if ( (fp = tmpfile()) == NULL) {      /* create temp */
    fprintf(stderr, "tmpfile create error: %s\n", strerror(errno));
    exit(1);
  }
  fputs("one line of output\n", fp);    /* write to temp */
  rewind(fp);                           /* then read back */
  if (fgets(line, sizeof(line), fp) == NULL) {
    fprintf(stderr, "i/o error: %s\n", strerror(errno)); exit(1);
  }
  fputs(line, stdout);          /* output line */

  exit(0);
}
```

Temporary Files Log

```
$ ./tmpfiles
/tmp/filea3SS9s
/tmp/fileLYsGQh
one line of output
$
```

References

Text, section 6.8 for setjmp/longjmp.

Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual*, 5th Edition, Prentice-Hall, 2002.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice-Hall, 1988.