

```

*****
12669 Fri May 2 20:34:24 2014
new/usr/src/cmd/printf/printf.c
4818 printf(1) should support n$ width and precision specifiers
*****
1 /*
2  * Copyright 2014 Garrett D'Amore <garrett@damore.org>
3  * Copyright 2010 Nexenta Systems, Inc. All rights reserved.
4  * Copyright (c) 1989, 1993
5  * The Regents of the University of California. All rights reserved.
6  *
7  * Redistribution and use in source and binary forms, with or without
8  * modification, are permitted provided that the following conditions
9  * are met:
10 * 1. Redistributions of source code must retain the above copyright
11 * notice, this list of conditions and the following disclaimer.
12 * 2. Redistributions in binary form must reproduce the above copyright
13 * notice, this list of conditions and the following disclaimer in the
14 * documentation and/or other materials provided with the distribution.
15 * 4. Neither the name of the University nor the names of its contributors
16 * may be used to endorse or promote products derived from this software
17 * without specific prior written permission.
18 *
19 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
20 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
21 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
22 * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
23 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
24 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
25 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
26 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
27 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
28 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
29 * SUCH DAMAGE.
30 */

32 #include <sys/types.h>

34 #include <err.h>
35 #include <errno.h>
36 #include <inttypes.h>
37 #include <limits.h>
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <string.h>
41 #include <unistd.h>
42 #include <alloca.h>
43 #include <ctype.h>
44 #include <locale.h>
45 #include <note.h>

47 #define warnx1(a, b, c) warnx(a)
48 #define warnx2(a, b, c) warnx(a, b)
49 #define warnx3(a, b, c) warnx(a, b, c)

51 #define PTRDIFF(x, y) ((uintptr_t)(x) - (uintptr_t)(y))

53 #define _(x) gettext(x)

55 #define PF(f, func) do { \
56     char *b = NULL; \
57     int dollar = 0; \
58     if (*f == '$') { \
59         dollar++; \
60         *f = '%'; \
61     } \
62 }

```

```

57     if (havewidth) \
58         if (haveprec) \
59             (void) asprintf(&b, f, fieldwidth, precision, func); \
60         else \
61             (void) asprintf(&b, f, fieldwidth, func); \
62     else if (haveprec) \
63         (void) asprintf(&b, f, precision, func); \
64     else \
65         (void) asprintf(&b, f, func); \
66     if (b) { \
67         (void) fputs(b, stdout); \
68         free(b); \
69     } \
70     if (dollar) \
71         *f = '$'; \
72     _NOTE(CONSTCOND) } while (0) \
73
72 static int  asciiicode(void);
73 static char *doformat(char *, int *);
74 static int  escape(char *, int, size_t *);
75 static int  getchr(void);
76 static int  getfloating(long double *, int);
77 static int  getint(int *);
78 static int  getnum(intmax_t *, uintmax_t *, int);
79 static const char
80     *getstr(void);
81 static char *mknum(char *, char);
82 static void  usage(void);

84 static const char digits[] = "0123456789";

86 static int  myargc;
87 static char **myargv;
88 static char **gargv;
89 static char **maxargv;

91 int
92 main(int argc, char *argv[])
93 {
94     size_t len;
95     int chopped, end, rval;
96     char *format, *fmt, *start;

98     (void) setlocale(LC_ALL, "");

100     argv++;
101     argc--;

103     /*
104     * POSIX says: Standard utilities that do not accept options,
105     * but that do accept operands, shall recognize "--" as a
106     * first argument to be discarded.
107     */
108     if (argc && strcmp(argv[0], "--") == 0) {
109         argc--;
110         argv++;
111     }

113     if (argc < 1) {
114         usage();
115         return (1);
116     }

118     /*
119     * Basic algorithm is to scan the format string for conversion
120     * specifications -- once one is found, find out if the field

```

```

121     * width or precision is a '*' ; if it is, gather up value. Note,
122     * format strings are reused as necessary to use up the provided
123     * arguments, arguments of zero/null string are provided to use
124     * up the format string.
125     */
126     fmt = format = *argv;
127     chopped = escape(fmt, 1, &len);          /* backslash interpretation */
128     rval = end = 0;
129     gargv = ++argv;

131     for (;;) {
132         maxargv = gargv;
133         char **maxargv = gargv;

134         myargv = gargv;
135         for (myargc = 0; gargv[myargc]; myargc++)
136             /* nop */;
137         start = fmt;
138         while (fmt < format + len) {
139             if (fmt[0] == '%') {
140                 (void) fwrite(start, 1, PTRDIFF(fmt, start),
141                     stdout);
142                 if (fmt[1] == '%') {
143                     /* %% prints a % */
144                     (void) putchar('%');
145                     fmt += 2;
146                 } else {
147                     fmt = doformat(fmt, &rval);
148                     if (fmt == NULL)
149                         return (1);
150                     end = 0;
151                 }
152                 start = fmt;
153             } else
154                 fmt++;
155             if (gargv > maxargv)
156                 maxargv = gargv;
157         }
158         gargv = maxargv;

160         if (end == 1) {
161             warnx1(_("missing format character"), NULL, NULL);
162             return (1);
163         }
164         (void) fwrite(start, 1, PTRDIFF(fmt, start), stdout);
165         if (chopped || !*gargv)
166             return (rval);
167         /* Restart at the beginning of the format string. */
168         fmt = format;
169         end = 1;
170     }
171     /* NOTREACHED */
172 }

175 static char *
176 doformat(char *fmt, int *rval)
177 doformat(char *start, int *rval)
178 {
179     static const char skip1[] = "#'-+ 0";
180     char **save;
181     static const char skip2[] = "0123456789";
182     char *fmt;
183     int fieldwidth, haveprec, havewidth, mod_ldbl, precision;
184     char convch, nextch;
185     char *start;

```

```

183     char *dptr;
184     int l;

186     start = alloca(strlen(fmt) + 1);
185     fmt = start + 1;

188     dptr = start;
189     *dptr++ = '%';
190     *dptr = 0;

192     fmt++;

194     /* look for "n$" field index specifier */
195     l = strspn(fmt, digits);
196     if ((l > 0) && (fmt[l] == '$')) {
197         int idx = atoi(fmt);
198         fmt += strspn(fmt, skip2);
199         if ((*fmt == '$') && (fmt != (start + 1))) {
200             int idx = atoi(start + 1);
201             if (idx <= myargc) {
202                 gargv = &myargv[idx - 1];
203             } else {
204                 gargv = &myargv[myargc];
205             }
206             if (gargv > maxargv) {
207                 maxargv = gargv;
208             }
209             fmt += l + 1;
210         }

211         /* skip to field width */
212         while (strchr(skip1, *fmt) != NULL) {
213             *dptr++ = *fmt++;
214             *dptr = 0;
215         }

217         if (*fmt == '*') {
218             start = fmt;
219             fmt++;
220             l = strspn(fmt, digits);
221             if ((l > 0) && (fmt[l] == '$')) {
222                 int idx = atoi(fmt);
223                 if (idx <= myargc) {
224                     gargv = &myargv[idx - 1];
225                 } else {
226                     gargv = &myargv[myargc];
227                 }
228                 fmt = start + 1;
229                 fmt += l + 1;
230             }

232             /* skip to field width */
233             fmt += strspn(fmt, skip1);
234             if (*fmt == '*') {
235                 if (getint(&fieldwidth))
236                     return (NULL);
237                 if (gargv > maxargv) {
238                     maxargv = gargv;
239                 }
240                 havewidth = 1;

242                 *dptr++ = '*';
243                 *dptr = 0;

```

```

208     ++fmt;
240 } else {
241     havewidth = 0;

243     /* skip to possible '.', get following precision */
244     while (isdigit(*fmt)) {
245         *dptr++ = *fmt++;
246         *dptr = 0;
213     fmt += strspn(fmt, skip2);
247     }
248 }

250 if (*fmt == '.') {
251     /* precision present? */
252     fmt++;
253     *dptr++ = '.';

217     ++fmt;
255     if (*fmt == '*') {

257         fmt++;
258         l = strspn(fmt, digits);
259         if ((l > 0) && (fmt[l] == '$')) {
260             int idx = atoi(fmt);
261             if (idx <= myargc) {
262                 gargv = &myargv[idx - 1];
263             } else {
264                 gargv = &myargv[myargc];
265             }
266             fmt += l + 1;
267         }

269         if (getint(&precision))
270             return (NULL);
271         if (gargv > maxargv) {
272             maxargv = gargv;
273         }
274         haveprec = 1;
275         *dptr++ = '*';
276         *dptr = 0;
222     ++fmt;
277 } else {
278     haveprec = 0;

280     /* skip to conversion char */
281     while (isdigit(*fmt)) {
282         *dptr++ = *fmt++;
283         *dptr = 0;
227     fmt += strspn(fmt, skip2);
284     }
285 }
286 } else
287     haveprec = 0;
288 if (!*fmt) {
289     warnx1(_("missing format character"), NULL, NULL);
290     return (NULL);
291 }
292 *dptr++ = *fmt;
293 *dptr = 0;

295 /*
296  * Look for a length modifier.  POSIX doesn't have these, so
297  * we only support them for floating-point conversions, which
298  * are extensions.  This is useful because the L modifier can
299  * be used to gain extra range and precision, while omitting
300  * it is more likely to produce consistent results on different

```

```

301     * architectures.  This is not so important for integers
302     * because overflow is the only bad thing that can happen to
303     * them, but consider the command printf %a 1.1
304     */
305     if (*fmt == 'L') {
306         mod_ldbl = 1;
307         fmt++;
308         if (!strchr("aAeEfFgG", *fmt)) {
309             warnx2(_("bad modifier L for %%%c"), *fmt, NULL);
310             return (NULL);
311         }
312     } else {
313         mod_ldbl = 0;
314     }

316     gargv = save;
317     convch = *fmt;
318     nextch = ++fmt;

320     *fmt = '\0';
321     switch (convch) {
322     case 'b': {
323         size_t len;
324         char *p;
325         int getout;

327         p = strdup(getstr());
328         if (p == NULL) {
329             warnx2("%s", strerror(ENOMEM), NULL);
330             return (NULL);
331         }
332         getout = escape(p, 0, &len);
333         *(fmt - 1) = 's';
334         PF(start, p);
335         *(fmt - 1) = 'b';
336         free(p);

338         if (getout)
339             return (fmt);
340         break;
341     }
342     case 'c': {
343         char p;

345         p = getch();
346         PF(start, p);
347         break;
348     }
349     case 's': {
350         const char *p;

352         p = getstr();
353         PF(start, p);
354         break;
355     }
356     case 'd': case 'i': case 'o': case 'u': case 'x': case 'X': {
357         char *f;
358         intmax_t val;
359         uintmax_t uval;
360         int signedconv;

362         signedconv = (convch == 'd' || convch == 'i');
363         if ((f = mknum(start, convch)) == NULL)
364             return (NULL);
365         if (getnum(&val, &uval, signedconv))
366             *rval = 1;

```

```
367         if (signedconv)
368             PF(f, val);
369         else
370             PF(f, uval);
371         break;
372     }
373     case 'e': case 'E':
374     case 'f': case 'F':
375     case 'g': case 'G':
376     case 'a': case 'A': {
377         long double p;
378
379         if (getfloating(&p, mod_ldbl))
380             *rval = 1;
381         if (mod_ldbl)
382             PF(start, p);
383         else
384             PF(start, (double)p);
385         break;
386     }
387     default:
388         warnx2(_("illegal format character %c"), convch, NULL);
389         return (NULL);
390     }
391     *fmt = nextch;
392     return (fmt);
393 }
```

unchanged_portion_omitted