****
12125 Mon Jul 15 08:41:48 2019
new/usr/src/lib/libdemangle/common/rust.c
11472 fix libdemangle rust changes
****
_____***unchanged_portion_omitted_***

```
220 static boolean_t
221 rustdem_parse_name_segment(rustdem_state_t *st, strview_t *svp, boolean_t first)
222 {
223         strview_t sv;
224         strview_t name;
225         uint64_t len;
226         size_t rem;
227         boolean_t last = B_FALSE;

229         if (st->rds_error != 0 || sv_remaining(svp) == 0)
230                 return (B_FALSE);

232         sv_init_sv(&sv, svp);

234         if (!rustdem_parse_num(st, &sv, &len)) {
235                 DEMDEBUG("ERROR: no leading length");
236                 st->rds_error = EINVAL;
237                 return (B_FALSE);
238         }

240         rem = sv_remaining(&sv);

242         if (rem < len) {
242         if (rem < len || len == SIZE_MAX) {
243                 st->rds_error = EINVAL;
244                 return (B_FALSE);
245         }

247         /* Is this the last segment before the terminating E? */
248         if (rem == len + 1) {
249                 VERIFY3U(sv_peek(&sv, -1), ==, 'E');
250                 last = B_TRUE;
251         }

253         if (!first && !rustdem_add_sep(st))
254                 return (B_FALSE);

256         /* Reduce length of seg to the length we parsed */
257         (void) sv_init_sv_range(&name, &sv, len);

259         DEMDEBUG("%s: segment='%.*s'", __func__, SV_PRINT(&name));

261         /*
262          * A rust hash starts with 'h', and is the last component of a name
263          * before the terminating 'E'
264          */
265         if (sv_peek(&name, 0) == 'h' && last) {
266                 if (!rustdem_parse_hash(st, &name))
267                         return (B_FALSE);
268                 goto done;
269         }

271         while (sv_remaining(&name) > 0) {
272                 switch (sv_peek(&name, 0)) {
273                 case '$':
274                         if (rustdem_parse_special(st, &name))
275                                 continue;
276                         break;
277                 case '_':
```



```
278                         if (sv_peek(&name, 1) == '$') {
279                                 /*
280                                  * Only consume/ignore '_'.  Leave
281                                  * $ for next round.
282                                  */
283                                 sv_consume_n(&name, 1);
284                                 continue;
285                         }
286                         break;
287                 case '.':
288                         /* Convert '..' to '::' */
289                         if (sv_peek(&name, 1) != '.')
290                                 break;

292                         if (!rustdem_add_sep(st))
293                                 return (B_FALSE);

295                         sv_consume_n(&name, 2);
296                         continue;
297                 default:
298                         break;
299                 }

301                 if (custr_appendc(st->rds_demangled,
302                     sv_consume_c(&name)) != 0) {
303                         st->rds_error = ENOMEM;
304                         return (B_FALSE);
305                 }
306         }

308 done:
309         DEMDEBUG("%s: consumed '%.*s'", __func__, (int)len, svp->sv_first);
310         sv_consume_n(&sv, len);
311         sv_init_sv(svp, &sv);
312         return (B_TRUE);
313 }
_____***unchanged_portion_omitted_***

386 /*
387  * We have to pick an arbitrary limit here; 999,999,999 fits comfortably
388  * within an int32_t, so let's go with that, as it seems unlikely we'd
389  * ever see a larger value in context.
387  * A 10 digit value would imply a name 1Gb or larger in size.  It seems
388  * unlikely to the point of absurdity any such value could every possibly
389  * be valid (or even have compiled properly).  This also prevents the
390  * uint64_t conversion from possibly overflowing since the value must always
391  * be below 10 * UINT32_MAX.
390  */
391 #define MAX_DIGITS 9
393 #define MAX_DIGITS 10

393 static boolean_t
394 rustdem_parse_num(rustdem_state_t *restrict st, strview_t *restrict svp,
395     uint64_t *restrict valp)
396 {
397         strview_t snum;
398         uint64_t v = 0;
399         size_t ndigits = 0;
400         char c;

402         if (st->rds_error != 0)
403                 return (B_FALSE);

405         sv_init_sv(&snum, svp);

407         DEMDEBUG("%s: str='%.*s'", __func__, SV_PRINT(&snum));
```

```
409         c = sv_peek(&snum, 0);
410         if (!ISDIGIT(c)) {
411                 DEMDEBUG("%s: ERROR no digits in str\n", __func__);
412                 st->rds_error = EINVAL;
413                 return (B_FALSE);
414         }

416         /*
417          * Since there is currently no official specification on rust name
418          * mangling, only that it has been stated that rust follows what
419          * C++ mangling does.  In the Itanium C++ ABI (what practically
420          * every non-Windows C++ implementation uses these days), it
421          * explicitly disallows leading 0s in numeric values (except for
422          * substition and template indexes, which aren't relevant here).
423          * We enforce the same restriction -- if a rust implementation allowed
424          * leading zeros in numbers (basically segment lengths) it'd
425          * cause all sorts of ambiguity problems with names that likely lead
426          * to much bigger problems with linking and such, so this seems
427          * reasonable.
428          */
429         if (c == '0') {
430                 DEMDEBUG("%s: ERROR number starts with leading 0\n", __func__);
431                 st->rds_error = EINVAL;
432                 return (B_FALSE);
433         }

435         while (sv_remaining(&snum) > 0 && ndigits <= MAX_DIGITS) {
436                 c = sv_consume_c(&snum);

438                 if (!ISDIGIT(c))
439                         break;

441                 v *= 10;
442                 v += c - '0';
443                 ndigits++;
444         }

446         if (ndigits > MAX_DIGITS) {
447                 DEMDEBUG("%s: value %llu is too large\n", __func__, v);
448                 st->rds_error = ERANGE;
449                 return (B_FALSE);
450         }

452         DEMDEBUG("%s: num=%llu", __func__, v);

454         *valp = v;
455         sv_consume_n(svp, ndigits);
456         return (B_TRUE);
457 }
```

_____***unchanged_portion_omitted_***