****
    3050 Thu Apr 2 12:57:05 2015
new/usr/src/test/libc-tests/cfg/symbols/README
2nd round review feedback from rmustacc.
****

```
#
# This file and its contents are supplied under the terms of the
# Common Development and Distribution License ("CDDL"), version 1.0.
# You may only use this file in accordance with the terms of version
# 1.0 of the CDDL.
#
# A full copy of the text of the CDDL should have accompanied this
# source.  A copy of the CDDL is also available via the Internet at
# http://www.illumos.org/license/CDDL.
#

#
# Copyright 2015 Garrett D'Amore <garrett@damore.org>
#

The configuration files in this directory are structured using the
syntax defined in the ../README file.  They make use of the compilation
environments declared in ../compilation.cfg, and are processed by the
symbols test.

We have organized the files by header file, that is the tests for symbols
declared in a header file (e.g. <unistd.h> appear in a file based on that
header file's name (e.g. unistd_h.cfg.)  This is purely for convenience.

Within these various declarations, we have the following field types:

<envs>    This is a list of compilation environments where the symbol
          should be legal.  To indicate that the symbol must not be legal
          an environment group can be prefixed with "-".  For example,
          "SUS+ -SUSv4+" indicates a symbol that is legal in all SUS
          "SUS -SUSv4+" indicates a symbol that is legal in all SUS
          environments up to SUSv3, and was removed in SUSv4 and subsequent
          versions of SUS.  As you can see, we can list multiple environments
          or environment groups, and we can add or remove to previous groups
          with subsequent ones.

<name>    This is a symbol name.  It follows the rules for C symbol names.

<header>  This is a header file, for example, unistd.h.  Conventionally,
          the header files used should match the file where the test is
          declared.

<type>    This is a C type.  Function types can be declared without their
          names, e.g. "void (*)(int)".  Structures (e.g. "struct stat") and
          pointer types (e.g. "pthead_t *") are legal as well.

Here are the types of declarations in these files:

type | <name> | <header> | <envs>

    Tests for a C type with <name>.  The test verifies that a variable with
    this type can be declared when the <header> is included.

value | <name> | <type> | <header> | <envs>

    Tests for a value named <name>, of type <type>.  The test attempts to
    assign the given value to a scratch variable declared with the given
    type.  The value can be a macro or other C symbol.

func | <name> | <type> | <type> [; <type> ]... | <header> | <envs>
```

```
    Tests whether a function <name>, returning the first <type>, and
    taking arguments of following <type> values, is declared.  Note that
    the argument types are separated by semicolons.  For varargs style
    functions, leave out the ... part.  For function declarations
    that have no declared arguments, either void can specified, or
    the type list can be omitted.

Examples:

    type | size_t | sys/types.h | ALL
    value | NULL | void * | stdlib.h | ALL
    func strnlen | int | const char *; int | string.h | ALL
```

```
**********************************************************
   18221 Thu Apr  2 12:57:06 2015
new/usr/src/test/libc-tests/tests/symbols/symbols_test.c
2nd round review feedback from rmustacc.
**********************************************************
______unchanged_portion_omitted_

char *compiler = NULL;
const char *c89flags = NULL;
const char *c99flags = NULL;

#define MAXENV  64      /* maximum number of environments (bitmask width) */
#define MAXHDR  10      /* maximum # headers to require to access symbol */
#define MAXARG  20      /* maximum # of arguments */

#define WS      " \t"

static int next_env = 0;

struct compile_env {
        char            *ce_name;
        char            *ce_lang;
        char            *ce_defs;
        int             ce_index;
        char            *name;
        char            *lang;
        char            *defs;
        int             index;
};

static struct compile_env compile_env[MAXENV];

struct env_group {
        char                    *eg_name;
        uint64_t                eg_mask;
        struct env_group        *eg_next;
        char                    *name;
        uint64_t                mask;
        struct env_group        *next;
};

typedef enum { SYM_TYPE, SYM_VALUE, SYM_FUNC } sym_type_t;

struct sym_test {
        char                    *st_name;
        sym_type_t              st_type;
        char                    *st_hdrs[MAXHDR];
        char                    *st_rtype;
        char                    *st_atypes[MAXARG];
        uint64_t                st_test_mask;
        uint64_t                st_need_mask;
        char                    *st_prog;
        struct sym_test         *st_next;
        char                    *name;
        sym_type_t              type;
        char                    *hdrs[MAXHDR];
        char                    *rtype;
        char                    *atypes[MAXARG];
        uint64_t                test_mask;
        uint64_t                need_mask;
        char                    *prog;
        struct sym_test         *next;
};
______unchanged_portion_omitted_

static void
```

```
append_sym_test(struct sym_test *st)
{
        *sym_insert = st;
        sym_insert = &st->st_next;
        sym_insert = &st->next;
}

static int
find_env_mask(const char *name, uint64_t *mask)
{
        for (int i = 0; i < MAXENV; i++) {
                if (compile_env[i].ce_name != NULL &&
                    strcmp(compile_env[i].ce_name, name) == 0) {
        for (int i = 0; i < 64; i++) {
                if (compile_env[i].name != NULL &&
                    strcmp(compile_env[i].name, name) == 0) {
                        *mask |= (1ULL << i);
                        return (0);
                }
        }

        for (struct env_group *eg = env_groups; eg != NULL; eg = eg->eg_next) {
                if (strcmp(name, eg->eg_name) == 0) {
                        *mask |= eg->eg_mask;
        for (struct env_group *eg = env_groups; eg != NULL; eg = eg->next) {
                if (strcmp(name, eg->name) == 0) {
                        *mask |= eg->mask;
                        return (0);
                }
        }
        return (-1);
}
______unchanged_portion_omitted_

static int
do_env(char **fields, int nfields, char **err)
{
        char *name;
        char *lang;
        char *defs;

        if (nfields != 3) {
                myasprintf(err, "number of fields (%d) != 3", nfields);
                return (-1);
        }

        if (next_env >= MAXENV) {
                myasprintf(err, "too many environments");
                return (-1);
        }

        name = fields[0];
        lang = fields[1];
        defs = fields[2];

        compile_env[next_env].ce_name = mystrdup(name);
        compile_env[next_env].ce_lang = mystrdup(lang);
        compile_env[next_env].ce_defs = mystrdup(defs);
        compile_env[next_env].ce_index = next_env;
        compile_env[next_env].name = mystrdup(name);
        compile_env[next_env].lang = mystrdup(lang);
        compile_env[next_env].defs = mystrdup(defs);
        compile_env[next_env].index = next_env;
        next_env++;
        return (0);
}
```

```
static int
do_env_group(char **fields, int nfields, char **err)
{
        char *name;
        char *list;
        struct env_group *eg;
        uint64_t mask;
        char *item;

        if (nfields != 2) {
                myasprintf(err, "number of fields (%d) != 2", nfields);
                return (-1);
        }

        name = fields[0];
        list = fields[1];
        mask = 0;

        if (expand_env(list, &mask, &item) < 0) {
                myasprintf(err, "reference to undefined env %s", item);
                return (-1);
        }

        eg = myzalloc(sizeof (*eg));
        eg->eg_name = mystrdup(name);
        eg->eg_mask = mask;
        eg->eg_next = env_groups;
        eg->name = mystrdup(name);
        eg->mask = mask;
        eg->next = env_groups;
        env_groups = eg;
        return (0);
}
______unchanged_portion_omitted_

static void
mkprog(struct sym_test *st)
{
        char *s;

        proglen = 0;

        for (int i = 0; i < MAXHDR && st->st_hdrs[i] != NULL; i++) {
                addprogfmt("#include <%s>\n", st->st_hdrs[i]);
        for (int i = 0; i < MAXHDR && st->hdrs[i] != NULL; i++) {
                addprogfmt("#include <%s>\n", st->hdrs[i]);
        }

        for (s = st->st_rtype; *s; s++) {
        for (s = st->rtype; *s; s++) {
                addprogch(*s);
                if (*s == ’(’) {
                        s++;
                        addprogch(*s);
                        s++;
                        break;
                }
        }
        addprogch(’ ’);

        /* for function pointers, s is closing suffix, otherwise empty */

        switch (st->st_type) {
        switch (st->type) {
        case SYM_TYPE:
```

```
                addprogstr("test_type;");
                break;

        case SYM_VALUE:
                addprogfmt("test_value%s;\n", s);       /* s usually empty */
                addprogstr("void\ntest_func(void)\n{\n");
                addprogfmt("\ttest_value = %s;\n}", st->st_name);
                addprogfmt("\ttest_value = %s;\n}", st->name);
                break;

        case SYM_FUNC:
                addprogstr("\ntest_func(");
                for (int i = 0; st->st_atypes[i] != NULL && i < MAXARG; i++) {
                for (int i = 0; st->atypes[i] != NULL && i < MAXARG; i++) {
                        int didname = 0;
                        if (i > 0) {
                                addprogstr(", ");
                        }
                        if (strcmp(st->st_atypes[i], "void") == 0) {
                        if (strcmp(st->atypes[i], "void") == 0) {
                                didname = 1;
                        }
                        if (strcmp(st->st_atypes[i], "") == 0) {
                        if (strcmp(st->atypes[i], "") == 0) {
                                didname = 1;
                                addprogstr("void");
                        }

                        /* print the argument list */
                        for (char *a = st->st_atypes[i]; *a; a++) {
                        for (char *a = st->atypes[i]; *a; a++) {
                                if (*a == ’(’ && a[1] == ’*’ && !didname) {
                                        addprogfmt("(*a%d", i);
                                        didname = 1;
                                        a++;
                                } else if (*a == ’[’ && !didname) {
                                        addprogfmt("a%d[", i);
                                        didname = 1;
                                } else {
                                        addprogch(*a);
                                }
                        }
                        if (!didname) {
                                addprogfmt(" a%d", i);
                        }
                }

                if (st->st_atypes[0] == NULL) {
                if (st->atypes[0] == NULL) {
                        addprogstr("void");
                }

                /*
                 * Close argument list, and closing ")" for func ptrs.
                 * Note that for non-function pointers, s will be empty
                 * below, otherwise it points to the trailing argument
                 * list.
                 */
                addprogfmt(")%s\n{\n\t", s);
                /* close argument list, and closing ")" for func ptrs */
                addprogfmt(")%s\n{\n\t", s);    /* NB: s is normally empty */

                if (strcmp(st->st_rtype, "") != 0 &&
                    strcmp(st->st_rtype, "void") != 0) {
                if (strcmp(st->rtype, "") != 0 &&
                    strcmp(st->rtype, "void") != 0) {
                        addprogstr("return ");
                }

                /* add the function call */
                addprogfmt("%s(", st->st_name);
                for (int i = 0; st->st_atypes[i] != NULL && i < MAXARG; i++) {
                        if (strcmp(st->st_atypes[i], "") != 0 &&
                            strcmp(st->st_atypes[i], "void") != 0) {
                addprogfmt("%s(", st->name);
                for (int i = 0; st->atypes[i] != NULL && i < MAXARG; i++) {
                        if (strcmp(st->atypes[i], "") != 0 &&
                            strcmp(st->atypes[i], "void") != 0) {
                                addprogfmt("%sa%d", i > 0 ? ", " : "", i);
                        }
                }

                addprogstr(");\n}");
                break;
        }

        addprogch('\n');

        st->st_prog = progbuf;
        st->prog = progbuf;
}

static int
add_envs(struct sym_test *st, char *envs, char **err)
{
        char *item;
        if (expand_env_list(envs, &st->st_test_mask, &st->st_need_mask,
            &item) < 0) {
        if (expand_env_list(envs, &st->test_mask, &st->need_mask, &item) < 0) {
                myasprintf(err, "bad env action %s", item);
                return (-1);
        }
        return (0);
}

static int
add_headers(struct sym_test *st, char *hdrs, char **err)
{
        int i = 0;

        for (char *h = strsep(&hdrs, ";"); h != NULL; h = strsep(&hdrs, ";")) {
                if (i >= MAXHDR) {
                        myasprintf(err, "too many headers");
                        return (-1);
                }
                test_trim(&h);
                st->st_hdrs[i++] = mystrdup(h);
                st->hdrs[i++] = mystrdup(h);
        }

        return (0);
}

static int
add_arg_types(struct sym_test *st, char *atype, char **err)
{
        int i = 0;
        char *a;
        for (a = strsep(&atype, ";"); a != NULL; a = strsep(&atype, ";")) {
                if (i >= MAXARG) {
                        myasprintf(err, "too many arguments");
                        return (-1);
```

```
                }
                test_trim(&a);
                st->st_atypes[i++] = mystrdup(a);
                st->atypes[i++] = mystrdup(a);
        }

        return (0);
}

static int
do_type(char **fields, int nfields, char **err)
{
        char *decl;
        char *hdrs;
        char *envs;
        struct sym_test *st;

        if (nfields != 3) {
                myasprintf(err, "number of fields (%d) != 3", nfields);
                return (-1);
        }
        decl = fields[0];
        hdrs = fields[1];
        envs = fields[2];

        st = myzalloc(sizeof (*st));
        st->st_type = SYM_TYPE;
        st->st_name = mystrdup(decl);
        st->st_rtype = mystrdup(decl);
        st->type = SYM_TYPE;
        st->name = mystrdup(decl);
        st->rtype = mystrdup(decl);

        if ((add_envs(st, envs, err) < 0) ||
            (add_headers(st, hdrs, err) < 0)) {
                return (-1);
        }
        append_sym_test(st);

        return (0);
}

static int
do_value(char **fields, int nfields, char **err)
{
        char *name;
        char *type;
        char *hdrs;
        char *envs;
        struct sym_test *st;

        if (nfields != 4) {
                myasprintf(err, "number of fields (%d) != 4", nfields);
                return (-1);
        }
        name = fields[0];
        type = fields[1];
        hdrs = fields[2];
        envs = fields[3];

        st = myzalloc(sizeof (*st));
        st->st_type = SYM_VALUE;
        st->st_name = mystrdup(name);
        st->st_rtype = mystrdup(type);
        st->type = SYM_VALUE;
        st->name = mystrdup(name);
        st->rtype = mystrdup(type);

        if ((add_envs(st, envs, err) < 0) ||
            (add_headers(st, hdrs, err) < 0)) {
                return (-1);
        }
        append_sym_test(st);

        return (0);
}

static int
do_func(char **fields, int nfields, char **err)
{
        char *name;
        char *rtype;
        char *atype;
        char *hdrs;
        char *envs;
        struct sym_test *st;

        if (nfields != 5) {
                myasprintf(err, "number of fields (%d) != 5", nfields);
                return (-1);
        }
        name = fields[0];
        rtype = fields[1];
        atype = fields[2];
        hdrs = fields[3];
        envs = fields[4];

        st = myzalloc(sizeof (*st));
        st->st_type = SYM_FUNC;
        st->st_name = mystrdup(name);
        st->st_rtype = mystrdup(rtype);
        st->type = SYM_FUNC;
        st->name = mystrdup(name);
        st->rtype = mystrdup(rtype);

        if ((add_envs(st, envs, err) < 0) ||
            (add_headers(st, hdrs, err) < 0) ||
            (add_arg_types(st, atype, err) < 0)) {
                return (-1);
        }
        append_sym_test(st);

        return (0);
}

struct sym_test *
next_sym_test(struct sym_test *st)
{
        return (st == NULL ? sym_tests : st->st_next);
        return (st == NULL ? sym_tests : st->next);
}

const char *
sym_test_prog(struct sym_test *st)
{
        if (st->st_prog == NULL) {
        if (st->prog == NULL) {
                mkprog(st);
        }
        return (st->st_prog);
        return (st->prog);
}
```

```
const char *
sym_test_name(struct sym_test *st)
{
        return (st->st_name);
        return (st->name);
}

/*
 * Iterate through tests.  Pass in NULL for cenv to begin the iteration. For
 * subsequent iterations, use the return value from the previous iteration.
 * Returns NULL when there are no more environments.
 */
struct compile_env *
sym_test_env(struct sym_test *st, struct compile_env *cenv, int *need)
{
        int i = cenv ? cenv->ce_index + 1: 0;
        int i = cenv ? cenv->index + 1: 0;
        uint64_t b = 1ULL << i;

        while ((i < MAXENV) && (b != 0)) {
                cenv = &compile_env[i];
                if (b & st->st_test_mask) {
                        *need = (st->st_need_mask & b) ? 1 : 0;
                if (b & st->test_mask) {
                        *need = (st->need_mask & b) ? 1 : 0;
                        return (cenv);
                }
                b <<= 1;
                i++;
        }
        return (NULL);
}

const char *
env_name(struct compile_env *cenv)
{
        return (cenv->ce_name);
        return (cenv->name);
}

const char *
env_lang(struct compile_env *cenv)
{
        return (cenv->ce_lang);
        return (cenv->lang);
}

const char *
env_defs(struct compile_env *cenv)
{
        return (cenv->ce_defs);
        return (cenv->defs);
}
______unchanged_portion_omitted_

void
find_compiler(void)
{
        test_t t;
        int i;
        FILE *cf;

        t = test_start("finding compiler");

        if ((cf = fopen(cfile, "w+")) == NULL) {
                test_failed(t, "Unable to open %s for write: %s", cfile,
                    strerror(errno));
                return;
        }
        (void) fprintf(cf, "#include <stdio.h>\n");
        (void) fprintf(cf, "int main(int argc, char **argv) {\n");
        (void) fprintf(cf, "#if defined(__SUNPRO_C)\n");
        (void) fprintf(cf, "exit(51);\n");
        (void) fprintf(cf, "#elif defined(__GNUC__)\n");
        (void) fprintf(cf, "exit(52);\n");
        (void) fprintf(cf, "#else\n");
        (void) fprintf(cf, "exit(99)\n");
        (void) fprintf(cf, "#endif\n}\n");
        (void) fclose(cf);

        for (i = 0; compilers[i] != NULL; i++) {
                char cmd[256];
                int rv;

                (void) snprintf(cmd, sizeof (cmd),
                    "%s %s %s -o %s >/dev/null 2>&1",
                    compilers[i], MFLAG, cfile, efile);
                test_debugf(t, "trying %s", cmd);
                rv = system(cmd);

                test_debugf(t, "result: %d", rv);

                if ((rv < 0) || !WIFEXITED(rv) || WEXITSTATUS(rv) != 0)
                        continue;

                rv = system(efile);
                if (rv >= 0 && WIFEXITED(rv)) {
                        rv = WEXITSTATUS(rv);
                } else {
                        rv = -1;
                }

                switch (rv) {
                case 51:        /* STUDIO */
                        test_debugf(t, "Found Studio C");
                        c89flags = "-Xc -errwarn=%all -v -xc99=%none " MFLAG;
                        c99flags = "-Xc -errwarn=%all -v -xc99=%all " MFLAG;
                        if (extra_debug) {
                                test_debugf(t, "c89flags: %s", c89flags);
                                test_debugf(t, "c99flags: %s", c99flags);
                        }
                        test_passed(t);
                        break;
                case 52:        /* GCC */
                        test_debugf(t, "Found GNU C");
                        c89flags = "-Wall -Werror -std=c89 " MFLAG;
                        c99flags = "-Wall -Werror -std=c99 " MFLAG;
                        if (extra_debug) {
                                test_debugf(t, "c89flags: %s", c89flags);
                                test_debugf(t, "c99flags: %s", c99flags);
                        }
                        test_passed(t);
                        break;
                case 99:
                        test_debugf(t, "Found unknown (unsupported) compiler");
                        continue;
                default:
                        continue;
                }
                myasprintf(&compiler, "%s", compilers[i]);
                test_debugf(t, "compiler: %s", compiler);
```

```
                        return;
                }
        test_failed(t, "No compiler found.");
}
```

______***unchanged_portion_omitted_***

```
void
test_compile(void)
{
        struct sym_test *st;
        struct compile_env *cenv;
        test_t t;
        int need;

        for (st = next_sym_test(NULL); st; st = next_sym_test(st)) {
                if ((sym != NULL) && strcmp(sym, sym_test_name(st))) {
                        continue;
                }
                /* XXX: we really want a sym_test_desc() */
                for (cenv = sym_test_env(st, NULL, &need);
                    cenv != NULL;
                    cenv = sym_test_env(st, cenv, &need)) {
                        t = test_start("%s : %c%s", sym_test_name(st),
                        t = test_start("%s : %c%s", st->name,
                            need ? '+' : '-', env_name(cenv));
                        if (do_compile(t, st, cenv, need) == 0) {
                                test_passed(t);
                        }
                }
        }

        if (full_count > 0) {
                test_summary();
        }
}
```

______***unchanged_portion_omitted_***