NAME

xstr – extract strings from C programs to implement shared strings

SYNOPSIS

xstr **–c** *filename* [**–v**] [**–l** *array*]
xstr [**–l** *array*]
xstr *filename* [**–v**] [**–l** *array*]

AVAILABILITY

SUNWcsu

DESCRIPTION

xstr maintains a file called **strings** into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, which are most useful if they are also read-only.

The command:

example% xstr –c *filename*

extracts the strings from the C source in name, replacing string references by expressions of the form **&xstr[***number***]** for some number. An appropriate declaration of **xstr** is prepended to the file. The resulting C text is placed in the file **x.c**, to then be compiled. The strings from this file are placed in the **strings** data base if they are not there already. Repeated strings and strings which are suffixes of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file declaring the common **xstr** space called **xs.c** can be created by a command of the form:

example% xstr

This **xs.c** file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

xstr can also be used on a single file. A command:

example% xstr *filename*

creates files **x.c** and **xs.c** as before, without using or affecting any **strings** file in the same directory.

It may be useful to run **xstr** after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. **xstr** reads from the standard input when the argument '–' is given. An appropriate command sequence for running **xstr** after the C preprocessor is:

example% cc –E name.c | xstr –c –
example% cc –c x.c
example% mv x.o name.o

xstr does not touch the file **strings** unless new items are added; thus **make**(1S) can avoid remaking **xs.o** unless truly necessary.

OPTIONS

–c *filename* Take C source text from *filename*.

–v Verbose: display a progress report indicating where new or duplicate strings were found.

–l *array* Specify the named *array* in program references to abstracted strings. The default array name is **xstr**.

FILES

strings	data base of strings
x.c	massaged C source
xs.c	C source for definition of array "xstr*(rq
/tmp/xs*	temp file when **xstr** *filename* doesn't touch **strings**

SEE ALSO

make(1S)

BUGS

If a string is a suffix of another string in the data base, but the shorter string is seen first by **xstr** both strings will be placed in the data base, when just placing the longer one there would do.

NOTES

Be aware that **xstr** indiscriminately replaces all strings with expressions of the form **&xstr[***number***]** regardless of the way the original C code might have used the string. For example, you will encounter a problem with code that uses **sizeof( )** to determine the length of a literal string because **xstr** will replace the literal string with a pointer that most likely will have a different size than the string's. To circumvent this problem:

- use **strlen( )** instead of **sizeof( )**; note that **sizeof( )** returns the size of the array (including the null byte at the end), whereas **strlen( )** doesn't count the null byte. The equivalent of **sizeof("***xxx***")** really is **(strlen("***xxx***"))+1**.
- use **#define** for operands of **sizeof( )** and use the **define**'d version. **xstr** ignores **#define** statements. Make sure you run **xstr** on *filename* before you run it on the preprocessor.

You will also encounter a problem when declaring an initialized character array of the form

char x[] = "*xxx***";**

xstr will replace *xxx* with an expression of the form **&xstr[***number***]** which will not compile. To circumvent this problem, use **static char *x = "xxx"** instead of **static char x[] = "xxx"**.