NAME

cproto – generate C function prototypes and convert function definitions

SYNOPSIS

cproto [*option ...*] [*file ...*]

DESCRIPTION

Extracting

Cproto generates function prototypes for functions defined in the specified C source files to the standard output. The function definitions may be in the old style or ANSI C style. Optionally, **cproto** also outputs declarations for variables defined in the files. If no *file* argument is given, **cproto** reads its input from the standard input.

Converting ANSI C

Cproto can convert function definitions in simple source files from the old style to the ANSI C style (see *BUGS* for limitations). The original source files along with files specified by

```
#include "file"
```

directives appearing in the source code will be overwritten with the converted code. If no file names are given on the command line, then the program reads the source code from the standard input and outputs the converted source to the standard output.

If any comments appear in the parameter declarations for a function definition, such as in the example,

```
main (argc, argv)
int argc;       /* number of arguments */
char *argv[];   /* arguments */
{
}
```

then the converted function definition will have the form

```
int
main (
    int argc,       /* number of arguments */
    char *argv[]   /* arguments */
)
{
}
```

Otherwise, the converted function definition will look like

```
int
main (int argc, char *argv[])
{
}
```

Converting K&R C

Cproto can convert function definitions from the ANSI style to the old K&R style. In this mode, the program also converts function declarators and prototypes that appear outside function bodies. This is not a complete ANSI C to old C conversion. The program does not change anything within function bodies.

Lint–libraries

Cproto can optionally generate source in lint–library format. This is useful in environments where the **lint** utility is used to supplement prototype checking of your program.

OPTIONS

–a Convert function definitions from the old style to the ANSI C style.

–B *directive*

Set the conditional compilation directive to output at the beginning of function definitions generated by the –b option. The default is

```
#ifdef ANSI_FUNC
```

−b Rewrite function definition heads to include both old style and new style declarations separated by a conditional compilation directive. For example, the program can generate this function definition:

```
#ifdef ANSI_FUNC

int
main (int argc, char *argv[])
#else

int
main (argc, argv)
int argc;
char *argv[]
#endif
{
}
```

−C *template*
Set the output format for function definitions with parameter comments. The −F and −P options allow setting the format for prototypes and function definitions.

The format is specified by a template in the form

```
" int f ( a, b )"
```

but you may replace each space in this string with any number of whitespace characters. For example, the option

```
−F"int f(\n\ta,\n\tb\n\t)"
```

will produce

```
int main(
        int argc,
        char *argv[]
        )
```

−c The parameter comments in the prototypes generated by the −f1 and −f2 options are omitted by default. Use this option to enable the output of these comments.

−D *name[=value]*
This option is passed through to the preprocessor and is used to define symbols for use with conditionals such as *#ifdef*.

−d Omit the definition of the prototype macro used by the −m option.

−E *cpp* Pipe the input files through the specified C preprocessor command when generating prototypes. By default, the program uses "/lib/cpp" via a pipe.

−E *0* Do not run the C preprocessor.

−e Output the keyword **extern** in front of every generated prototype or declaration that has global scope.

−F *template*
Set the output format for function definitions as in the "−C" option.

−f *n* Set the style of generated function prototypes where *n* is a number from 0 to 3. For example, consider the function definition

```
main (argc, argv)
int argc;
```

```
char *argv[];
{
}
```

If the value is 0, then no prototypes are generated. When set to 1, the output is:

```
int main(/*int argc, char *argv[]*/);
```

For a value of 2, the output has the form:

```
int main(int /*argc*/, char */*argv*/[]);
```

The default value is 3. It produces the full function prototype:

```
int main(int argc, char *argv[]);
```

−I *directory*
This option is passed through to the preprocessor and is used to specify a directory to search for files that are referenced with *#include*.

−i By default, **cproto** only generates declarations for functions and variables having global scope. This option will output **inline** declarations as well.

−l Generate text for a lint−library (overrides the "**−f**" option). The output includes the comment

```
/* LINTLIBRARY */
```

Special comments LINT_EXTERN and LINT_PREPRO (a la "VARARGS") respectively turn on the "−x" option and copy comment−text to the output (for preprocessing in **lint**). Use the comment

```
/* LINT_EXTERN2 */
```

to include externs defined in the first level of include−files. Use the comment

```
/* LINT_SHADOWED */
```

to cause **cproto** to put "#undef" directives before each lint library declaration (i.e., to avoid conflicts with macros that happen to have to have the same name as the functions, thus causing syntax errors).

Note that these special comments are not supported under VAX/VMS, since there is no equivalent for the "−C" option of the C preprocessor with VAX−C.

−M *name*
Set the name of the macro used to surround prototype parameter lists when option −m is selected. The default is "P_".

−m Put a macro around the parameter list of every generated prototype. For example:

```
int main P_((int argc, char *argv[]));
```

−N *name*
Rather than filling in "void" for functions without parameters, use the given *name*.

−n Rather than filling in "void" for functions without parameters, use a comment "/*empty*/".

−O *file* Specify the name of the error file (default: standard error).

−o *file* Specify the name of the output file (default: standard output).

−P *template*
Set the output format for generated prototypes as in the "−C" option.

−p Disable promotion of formal parameters in old style function definitions. By default, parameters of type **char** or **short** in old style function definitions are promoted to type **int** in the function prototype or converted ANSI C function definition. Parameters of type **float** get promoted to **double** as well.

−q Do not output any error messages when the program cannot read the file specified in an *#include* directive.

−S Output only static declarations.

−s By default, **cproto** only generates declarations for functions and variables having global scope. This option will output **static** declarations as well.

−T Copy type definitions from each file. (Definitions in included−files are copied, unlike the "−l" option).

−t Convert function definitions from the ANSI C style to the traditional style.

−U *name*
This option is passed through to the preprocessor and is used to remove any definitions of this symbol.

−V Print version information.

−v Also output declarations for variables defined in the source.

−X *level*
This option limits the include−file level from which declarations are extracted by examining the preprocessor output.

−x This option causes procedures and variables which are declared "extern" to be included in the output.

ENVIRONMENT

The environment variable CPROTO is scanned for a list of options in the same format as the command line options. Options given on the command line override any corresponding environment option.

BUGS

If an un−tagged struct, union or enum declaration appears in a generated function prototype or converted function definition, the content of the declaration between the braces is empty.

The program does not pipe the source files through the C preprocessor when it is converting function definitions (i.e., options −a, −b, −t). That is because **cproto** does the conversion by replacing strings, rather than by converting to/from lexical tokens. Instead of using the C preprocessor, **cproto** tries to handle preprocessor directives and macros itself, to find type definitions. There are several limitations as a result:

- the search-path for included files contains only "/usr/include" and the directories listed by "-I" options.
- conditional directives (e.g., "#if") are ignored.
- only macros without parameters are handled.
- some comments in the function definition head are discarded.

The −v option does not generate declarations for variables defined with the **extern** specifier. This does not strictly conform to the C language standard but this rule was implemented because include files commonly declare variables this way.

When the program encounters an error, it usually outputs the not very descriptive message "syntax error". (Your configuration may allow the extended error reporting in yyerror.c).

Options that take string arguments only interpret the following character escape sequences:

```
\n    newline
\s    space
\t    tab
```

VARARGS comments are not passed through on systems whose C preprocessors do not support this (e.g., VAX/VMS, MS−DOS).

AUTHORS

Chin Huang
cthuang@vex.net

cthuang@interlog.com

Thomas E. Dickey
dickey@invisible−island.net
modifications to support lint library, type−copying, and port to VAX/VMS.

SEE ALSO

cc(1), cpp(1)