c++decl(1) - Linux man page

Name

cdecl, c++decl - Compose C and C++ type declarations

Synopsis

cdecl [-a | -+ | -p | -r] [-ciqdDV]

[[*files* ...] | **explain** ... | **declare** ... | **cast** ... | **set** ... | **help** | **?**]

c++decl [-a | -+ | -p | -r] [-ciqdDV]

[[*files* ...] | **explain** ... | **declare** ... | **cast** ... | **set** ... | **help** | **?**]

explain ...
declare ...
cast ...

Description

Cdecl (and *c++decl*) is a program for encoding and decoding C (or C++) type declarations. The C language is based on the (draft proposed) X3J11 ANSI Standard; optionally, the C language may be based on the pre-ANSI definition defined by Kernighan & Ritchie's *The C Programming Language* book, or the C language defined by the Ritchie PDP-11 C compiler. The C++ language is based on Bjarne Stroustrup's *The C++ Programming Language*, plus the version 2.0 additions to the language.

Options

-a
: Use the ANSI C dialect of the C language.

-p
: Use the pre-ANSI dialect defined by Kernighan & Ritchie's book.

-r
: Use the dialect defined by the Ritchie PDP-11 C compiler.

-+
: Use the C++ language, rather than C.

-i
: Run in interactive mode (the default when reading from a terminal). This also turns on prompting, line editing, and line history.

-q
: Quiet the prompt. Turns off the prompt in interactive mode.

-c

Create compilable C or C++ code as output. *Cdecl* will add a semicolon to the end of a declaration and a pair of curly braces to the end of a function definition.

-d

Turn on debugging information (if compiled in).

-D

Turn on YACC debugging information (if compiled in).

-V

Display version information and exit.

Invoking

Cdecl may be invoked under a number of different names (by either renaming the executable, or creating a symlink or hard link to it). If it is invoked as *cdecl* then ANSI C is the default language. If it is invoked as *c++decl* then C++ is the default. If it is invoked as either *explain*, *cast*, or *declare* then it will interpret the rest of the command line options as parameters to that command, execute the command, and exit. It will also do this if the first non-switch argument on the command line is one of those three commands. Input may also come from a file.

Cdecl reads the named files for statements in the language described below. A transformation is made from that language to C (C++) or pseudo-English. The results of this transformation are written on standard output. If no files are named, or a filename of ''-'' is encountered, standard input will be read. If standard input is coming from a terminal, (or the **-i** option is used), a prompt will be written to the terminal before each line. The prompt can be turned off by the **-q** option (or the *set* noprompt command). If *cdecl* is invoked as *explain*, *declare* or *cast*, or the first argument is one of the commands discussed below, the argument list will be interpreted according to the grammar shown below instead of as file names.

When it is run interactively, *cdecl* uses the GNU readline library to provide keyword completion and command line history, very much like ***bash****(1)* (q.v.). Pressing TAB will complete the partial keyword before the cursor, unless there is more than one possible completion, in which case a second TAB will show the list of possible completions and redisplay the command line. The left and right arrow keys and backspace can be used for editing in a natural way, and the up and down arrow keys retrieve previous command lines from the history. Most other familiar keys, such as Ctrl-U to delete all text from the cursor back to the beginning of the line, work as expected. There is an ambiguity between the *int* and *into* keywords, but *cdecl* will guess which one you meant, and it always guesses correctly.

You can use *cdecl* as you create a C program with an editor like ***vi****(1)* or ***emacs****(1)*. You simply type in the pseudo-English version of the declaration and apply *cdecl* as a filter to the line. (In ***vi****(1)*, type ''**!!cdecl**<cr>''.)

If the *create program* option **-c** is used, the output will include semi-colons after variable declarations and curly brace pairs after function declarations.

The **-V** option will print out the version numbers of the files used to create the process. If the source is compiled with debugging information turned on, the **-d** option will enable it to be output. If the source is compiled with YACC debugging information turned on, the **-D** option will enable it to be output.

Command Language

There are six statements in the language. The *declare* statement composes a C type declaration from a verbose description. The *cast* statement composes a C type cast as might appear in an expression. The *explain* statement decodes a C type declaration or cast, producing a verbose description. The *help* (or *?*) statement provides a help message. The *quit* (or *exit*) statement (or the end of file) exits the program. The *set* statement allows the command line options to be set interactively. Each statement is separated by a semi-colon or a newline.

Synonyms

Some synonyms are permitted during a declaration:

```
      character   is a synonym for   char
       constant   is a synonym for   const
    enumeration   is a synonym for   enum
           func   is a synonym for   function
        integer   is a synonym for   int
            ptr   is a synonym for   pointer
            ref   is a synonym for   reference
            ret   is a synonym for   returning
      structure   is a synonym for   struct
         vector   is a synonym for   array
```

The TAB completion feature only knows about the keywords in the right column of the structure, not the ones in the left column. TAB completion is a lot less useful when the leading characters of different keywords are the same (the keywords confict with one another), and putting both columns in would cause quite a few conflicts.

Grammar

The following grammar describes the language. In the grammar, words in "<>" are non-terminals, bare lower-case words are terminals that stand for themselves. Bare upper-case words are other lexical tokens: NOTHING means the empty string; NAME means a C identifier; NUMBER means a string of decimal digits; and NL means the new-line or semi-colon characters.

```
<program>    ::= NOTHING
    | <program> <stmt> NL
<stmt>    ::= NOTHING
    | declare NAME as <adecl>
    | declare <adecl>
    | cast NAME into <adecl>
    | cast <adecl>
    | explain <optstorage> <ptrmodlist> <type> <cdecl>
    | explain <storage> <ptrmodlist> <cdecl>
    | explain ( <ptrmodlist> <type> <cast> ) optional-NAME
    | set <options>
    | help | ?
    | quit
    | exit
<adecl>    ::= array of <adecl>
    | array NUMBER of <adecl>
    | function returning <adecl>
    | function ( <adecl-list> ) returning <adecl>
    | <ptrmodlist> pointer to <adecl>
    | <ptrmodlist> pointer to member of class NAME <adecl>
    | <ptrmodlist> reference to <adecl>
    | <ptrmodlist> <type>
<cdecl>    ::= <cdecl1>
    | * <ptrmodlist> <cdecl>
    | NAME :: * <cdecl>
    | & <ptrmodlist> <cdecl>
<cdecl1>    ::= <cdecl1> ( )
    | <cdecl1> ( <castlist> )
    | <cdecl1> [ ]
    | <cdecl1> [ NUMBER ]
    | ( <cdecl> )
    | NAME
<cast>    ::= NOTHING
    | ( )
    | ( <cast> ) ( )
    | ( <cast> ) ( <castlist> )
    | ( <cast> )
    | NAME :: * <cast>
    | * <cast>
    | & <cast>
    | <cast> [ ]
    | <cast> [ NUMBER ]
```

```
<type>    ::= <typename> | <modlist>
    | <modlist> <typename>
    | struct NAME | union NAME | enum NAME | class NAME
<castlist>    ::= <castlist> , <castlist>
    | <ptrmodlist> <type> <cast>
    | <name>
<adecllist>    ::= <adecllist> , <adecllist>
    | NOTHING
    | <name>
    | <adecl>
    | <name> as <adecl>
<typename>    ::= int | char | double | float | void
<modlist>    ::= <modifier> | <modlist> <modifier>
<modifier>    ::= short | long | unsigned | signed | <ptrmod>
<ptrmodlist>    ::= <ptrmod> <ptrmodlist> | NOTHING
<ptrmod>    ::= const | volatile | noalias
<storage>    ::= auto | extern | register | auto
<optstorage>    ::= NOTHING | <storage>
<options>    ::= NOTHING | <options>
    | create | nocreate
    | prompt | noprompt
    | ritchie | preansi | ansi | cplusplus
    | debug | nodebug | yydebug | noyydebug
```

Set Options

The *set* command takes several options. You can type *set* or *set options* to see the currently selected options and a summary of the options which are available. The first four correspond to the *-a*, *-p*, *-r*, and -+ command line options, respectively.

ansi
: Use the ANSI C dialect of the C language.

preansi
: Use the pre-ANSI dialect defined by Kernighan & Ritchie's book.

ritchie
: Use the dialect defined by the Ritchie PDP-11 C compiler.

cplusplus
: Use the C++ language, rather than C.

[no]prompt
: Turn on or off the prompt in interactive mode.

[no]create
: Turn on or off the appending of semicolon or curly braces to the declarations output by *cdecl*. This corresponds to the *-c* command line option.

[no]debug
Turn on or off debugging information.
[no]yydebug
Turn on or off YACC debugging information.

Note: debugging information and YACC debugging information are only available if they have been compiled into *cdecl*. The last two options correspond to the *-d* and *-D* command line options, respectively. Debugging information is normally used in program development, and is not generally compiled into distributed executables.

Examples

To declare an array of pointers to functions that are like ***malloc****(3)*, do

declare fptab as array of pointer to function returning pointer to char

The result of this command is

char *(*fptab[])()

When you see this declaration in someone else's code, you can make sense out of it by doing

explain char *(*fptab[])()

The proper declaration for ***signal****(2)*, ignoring function prototypes, is easily described in *cdecl*'s language:

declare signal as function returning pointer to function returning void

which produces

void (*signal())()

The function declaration that results has two sets of empty parentheses. The author of such a function might wonder where to put the parameters:

declare signal as function (arg1,arg2) returning pointer to function returning void

provides the following solution (when run with the *-c* option):

void (*signal(arg1,arg2))() { }

If we want to add in the function prototypes, the function prototype for a function such as **_exit**(2) would be declared with:

declare _exit as function (retvalue as int) returning void

giving

void _exit(int retvalue) { }

As a more complex example using function prototypes, ***signal****(2)* could be fully defined as:

declare signal as function(x as int, y as pointer to function(int) returning void) returning pointer to function(int) returning void

giving (with -c)

void (*signal(int x, void (*y)(int)))(int) { }

Cdecl can help figure out the where to put the "const" and "volatile" modifiers in declarations, thus

declare foo as pointer to const int

gives

const int *foo

while

declare foo as const pointer to int

gives

int * const foo

C++decl can help with declaring references, thus

declare x as reference to pointer to character

gives

char *&x

C++decl can help with pointers to member of classes, thus declaring a pointer to an integer member of a class X with

declare foo as pointer to member of class X int

gives

int X::*foo

and

> declare foo as pointer to member of class X function (arg1, arg2) returning pointer to class Y

gives

> class Y *(X::*foo)(arg1, arg2)

Diagnostics

The declare, cast and explain statements try to point out constructions that are not supported in C. In some cases, a guess is made as to what was really intended. In these cases, the C result is a toy declaration whose semantics will work only in Algol-68. The list of unsupported C constructs is dependent on which version of the C language is being used (see the ANSI, pre-ANSI, and Ritchie options). The set of supported C++ constructs is a superset of the ANSI set, with the exception of the **noalias** keyword.

References

ANSI Standard X3.159-1989 (ANSI C)

ISO/IEC 9899:1990 (the ISO standard)

The comp.lang.c FAQ
http://www.eskimo.com/~scs/C-faq.top.html

Section 8.4 of the C Reference Manual within *The C Programming Language* by B. Kernighan & D. Ritchie.

Section 8 of the C++ Reference Manual within *The C++ Programming Language* by B. Stroustrup.

Caveats

The pseudo-English syntax is excessively verbose.

There is a wealth of semantic checking that isn't being done.

Cdecl was written before the ANSI C standard was completed, and no attempt has been made to bring it up-to-date. Nevertheless, it is very close to the standard, with the obvious exception of *noalias*.

Cdecl's scope is intentionally small. It doesn't help you figure out initializations. It expects storage classes to be at the beginning of a declaration, followed by the the const, volatile and noalias modifiers, followed by the type of the variable. *Cdecl* doesn't know anything about variable length argument lists. (This includes the ''*,...*'' syntax.)

Cdecl thinks all the declarations you utter are going to be used as external definitions. Some declaration contexts in C allow more flexibility than this. An example of this is:

> declare argv as array of array of char

where *cdecl* responds with

```
Warning: Unsupported in C -- 'Inner array of unspecified size'
        (maybe you mean "array of pointer")
char argv[][]
```

Tentative support for the *noalias* keyword was put in because it was in the draft ANSI specifications.

Authors

Originally written by Graham Ross, improved and expanded by David Wolverton, Tony Hansen, and Merlyn LeRoy.

GNU readline support and Linux port by David R. Conrad, <conrad@detroit.freenet.org>

See Also

bash(1), ***emacs***(1), ***malloc***(3), ***vi***(1).