

parse_conf library 1.2

The Library for Parsing Configuration Files with References Resolution

Anton Kulchitsky (anton@kulchitsky.org)

This manual is for `parse_conf` library (version 1.2, 30 March 2010).

`parse_conf` is a library for parsing standard configuration or initialization files and supports some extensions over the standard syntax. It supports the separation of config files into different groups and reading strings and numbers. The library is written in ANSI C and designed to be fine with C++. Originally (in 2002, 2004) it was designed for reading initial values for scientific applications of special type. It was rewritten and extended to be a flexible general purpose library. Since version 1.0, a very extensive reference substitution system is included into the library. The library is designed to be thread-safety and you can read/write as many configuration files simultaneously as you would like.

The *configuration file* consists of sections, led by a `[section]` header and followed by `name: value` entries, `name=value` is also accepted. You also can concatenate a string to the previous value using `name += value` or `name += value` commands. Note that leading and ending whitespaces are removed from string values and ignored from numeric values. To add a whitespace to a string, the string should be quoted by quote " sign or you may use `\s` and `\t` values for spaces and tabulations. The rest of lines from `#` or `;` are ignored and may be used to provide comments.

Any variable definition can contain references to other values. See documentation for the details. For example, to substitute the value from variable `a` of the same group or from the global variables, one may use `$a` reference or Pythonish `%(a)s`.

Copyright © 2004–2010 Anton Kulchitsky

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is not included in the manual itself. The electron version of this manual must be distributed with the file containing the full text of the license except when distributing the manual in html, pdf or ps formats.

Table of Contents

1	Introduction	1
2	Installation	2
3	Configuration File Format	3
3.1	Format Description	3
3.1.1	Sections	3
3.1.2	Fields	3
3.1.3	Comments	4
3.2	References	4
3.2.1	Local References	4
3.2.2	Global References	4
3.2.3	Characters \$, %	4
3.2.4	References Substitution Rule	4
3.3	List of Values	5
3.4	Quoted Format	5
3.5	Continuing Long Lines	6
3.6	Configuration Examples	6
4	Interface	8
4.1	Include Instruction	8
4.2	Handler Allocation	8
4.3	Flags	8
4.4	Configuration Reading	9
4.5	Getting Field Values	9
4.6	Getting Arrays	10
4.6.1	Getting Number of Elements	10
4.6.2	Getting Values	10
4.6.3	Array Usage Example	11
4.7	Getting Lists of Values	12
4.7.1	List Types	12
4.7.2	Allocation Lists	12
4.7.3	Getting Variables	12
4.7.4	Usage Example	13
4.8	Setting Field Values	14
4.9	Printing Configuration	14
4.10	Cleaning Configuration	14
4.11	Resolving References	15
4.12	Data Direct Access	15
4.13	Section Listing	16
5	Compiling and Linking	17

6	pcnfdump Utility	18
Appendix A	Index	19

1 Introduction

`parse_conf` is a library for parsing standard configuration or initialization files and supports some extensions over the standard syntax. It supports the separation of config files into different groups and reading strings and numbers. The library is written in ANSI C and designed to be fine with C++. Originally (in 2002, 2004) it was designed for reading initial values for scientific applications of special type. It was rewritten and extended to be a flexible general purpose library. Since version 1.0, a very extensive reference substitution system is included into the library. The library is designed to be thread-safety and you can read/write as many configuration files simultaneously as you would like.

The *configuration file* consists of sections, led by a `[section]` header and followed by `name: value` entries, `name=value` is also accepted. You also can concatenate a string to the previous value using `name += value` or `name +=: value` commands. Note that leading and ending whitespaces are removed from string values and ignored from numeric values. To add a whitespace to a string, the string should be quoted by quote " sign or you may use `\s` and `\t` values for spaces and tabulations. The rest of lines from `#` or `;` are ignored and may be used to provide comments.

Any variable definition can contain references to other values. See documentation for the details. For example, to substitute the value from variable `a` of the same group or from the global variables, one may use `$a` reference or Pythonish `%(a)s`.

2 Installation

To install the library and penfdump utility to a standard location, use

```
./configure
make
su -c "make install"
```

In the case you installing it locally, use

```
./configure --prefix=/desired/location
make
make install
```

instead. Do not forget to change `LIBRARY_PATH` and `C_INCLUDE_PATH` accordingly to be able to reach the library. You can also generate tests if you wish by using `--enable-tests` option for `configure`.

3 Configuration File Format

3.1 Format Description

`parse_conf` library reads a very common format for configuration or initialization files that is widely used. However, many important extensions are also supported. One of such an extension is a reference substitutions feature. The configuration file consists of *sections* or *groups* with variable string definitions (fields) that belong to these sections. You can make references to other fields and the library will substitute the values. You can reference to the variables defined anywhere else in the configuration file.

Another important extension to the format is a support for lists of values. Every field can contain a list of values separated by any single-character separator (it can be comma or vertical line, as an example). The lists of strings, integers, or doubles are supported.

3.1.1 Sections

Sections start with a *section declaration* like `[section name]` on a separate line. Such a line must contain only this declaration and possibly comments after it. All spaces around the declaration are ignored. All spaces around the name of the section are also ignored. Thus, `[section name]` is a valid declaration equivalent to that does not have spaces around the name, that is “section name”. Section names can contain spaces between words and they are meaningful. Section names are also case sensitive. Every section declaration ends the previous section. More formally, the section name must start with an alpha-numeric symbol, underscore or dash and must end with an alpha-numeric symbol, underscore or dash. Other symbols can contain everything except a new line, (,), [,], :, and =.

The *global section* is the only section that does not have a section declaration. Every variable defined before the first section declaration belongs to that global section (known also as a *NULL section*). Every configuration file contains at least one global section which may have no field in it. Thus, no section declaration are necessary in the configuration file. In such a file without section declarations, all variables belong to the global section.

3.1.2 Fields

All other lines can be used to read information from the configuration file. These lines contain either signs `:` or `=` as a separator of the variable (field name) and its definition line. It is possible to have more `:` or `=` in the line which are considered as a part of the definition line. The field also can use an increment sign (either `+:` or `+=`). If used, the increment sign will add the string to its previous value (concatenation) or will be treated as a simple initialization if there were no such field was defined before. Spaces around the definition line are ignored.

There are in the line that are treated differently than usual signs. First of all, the backslash `\` is a control sign that changes the meaning of the succeeding character. To type a `\` itself one need to use `\\`. `\s` represents a space and `\t` represents a tabulation. These signs are useful if definition line must contain spaces at the beginning or the end due to spaces around the definition line are ignored. Any appearance of `\"` are replaced by `"` in the definition lines. This may be useful if your definition line starts with `"`. You cannot use just this double quote `"` at the beginning of the line due to it will switch the input mode to

quoted definition lines. See [Section 3.4 \[Quoted Format\], page 5](#). `\` at the end of the line (with possible spaces or comments followed) indicates that the definition will continue at the next line. It is used to split long lines. `\#` stays for `#`, and `\;` stays for `;`.

3.1.3 Comments

Comments are any lines started with `#` or `;`, or the rest of any line from this sign unless (1) these signs appear inside the quoted strings See [Section 3.4 \[Quoted Format\], page 5](#), or (2) these signs appear in a regular string and preceded by backslash `\`. See [\[special sequences\], page 3](#). Comments can appear in any part of your configuration file. Any lines besides the comments, section names, and fields are considered to be a syntax error. These lines will cause your program to exit with an error message.

Field name or *variable name* is the part of a string before first `=`, `:`, `+=`, or `+`: without leading and trailing spaces. That is all leading and finishing spaces are deleted from the line. The name convention for field names is the same as for section names. See [Section 3.1.1 \[Sections\], page 3](#).

3.2 References

Definition lines of the fields can contains references to other fields. In this case the library will substitute the value of referenced field instead of the reference.

3.2.1 Local References

To make a reference to a field `x` defined in the same section one needs to use one of the following format: `$x`, `$(x)`, or `%(x)s`. All these references will first search for `x` definition in the same section. If not found, the search will be made in the global section. The last format is provided to make the library capable to read configuration files provided for `ConfigParser` Python module. The important difference from most of other libraries is that the references can reference to the fields defined later in the file.

You cannot use simple reference like `$x` if the name of the field in the reference contains spaces or any characters besides alpha-numeric, `-`, or `_`. For example to make a reference to the field `Three Words Field` one can use `$(Three Words Field)` or `%(Three Words Field)s` references only.

3.2.2 Global References

You can make a reference from one section to another. The form of such a reference to a variable `x` in a section `y` can be `[$y]x` or `[$y](x)`. The last form is suitable when both a field name and a group name contain spaces in them. To make a reference to the global section, one may use `[$]x` or `[$](x)` form. In this case `x` will not be searched in the local section first.

3.2.3 Characters \$, %

One need to use a special sequences to be able to use `$`, and `%` in their configuration files. They are `\\$`, `\\%` correspondingly. Please notice two backslashes, not one. It is because of the References Substitution Rule. The first parsing replaces two backslashes by one.

3.2.4 References Substitution Rule

One may ask what would happen if the field has been changed a few times in the file. To understand what happens in this case, one should understand how substitutions work. The parsing of the configuration file contains two independent stages. First, the configuration file is read without any substitutions and all references are treated as a plain text. This may be the last stage if you wish and use special flags for your library. See [Chapter 4 \[Interface\]](#), page 8. However, there may be the second stage of substitutions.

After the file initially parsed, every definition line is parsed for references. However, it may be not the same definition line as it was in the file. Thus, if you have a few definitions or increments of the same field, then the combined results will be parsed for references. Suppose you have in you ini-file:

```
field1 = Hello $x,  
field1 += \sBye $x.
```

Then the definition line to be parsed will be “Hello \$x, Bye \$x”.

At the second stage the resolution does not care when the variable was defined. It parses and makes substitutions for the resulted definitions. The referenced fields can contain other references as well. The only evident exception is that no recursive references are allowed. If they occur, the library will notify about the error and will not resolve that references.

3.3 List of Values

Since version 1.1, every field can contain a list of values instead of a single value. Technically, it is not a format change, due to it is up to the programmer how to treat any single field: as a single value or as a list using special list type and interface See [Section 4.7 \[Getting Lists of Values\]](#), page 12. Since version 1.2, a simplified interface to read list of doubles and integers into arrays is added See [Section 4.6 \[Getting Arrays\]](#), page 10. Array interface is much simpler than list interface but can be used only for numbers.

List of values contains values separated with a single character separator (like comma, column etc.). It may contain just one value or no value at all. All reference rules are applied to the list as well as to a single value. If the list is referenced from another field, the whole list is substituted as a string value to the reference. Programmer uses a special interface to treat the field value as a list. Programmer also specifies any separator which can change from one field to another if desired.

Lists of integers and doubles are supported as well as a most general list of strings.

3.4 Quoted Format

Quotes can be used in the field definition lines as below:

```
a = "This is a quoted definition line"
```

If such a form is used, other special characters are defined. #, ; can be used without \. In addition \" for a quote is necessary. It is defined in a standard format as well but need to be used there only if the string starts from a quote.

3.5 Continuing Long Lines

Continuing long lines is easy, just put a backslash `\` at the end of the definition line. You can put comments and spaces after this backslash but nothing else. The following long definition is all considered on one line:

```
SRC_FILES = application.cc \    # files to compile
filelister.cc menu\ #        # a word split!
item.cc \
usepix.cc
```

Thus the field `SRC_FILES` will be defined as “application.cc filelister.cc menuitem.cc usepix.cc”.

3.6 Configuration Examples

There is an example of simple initialization file:

```
default group string: "just a string"

[ INITIAL CONDITIONS ]

# velocity
v = 100.5 # km/s
n = 5.1   # 1/m^3

[ DATE ] # the date of the event

# What month is to be output (just a number, f.e. 4 is April)
Month = 3 #this value will be read

[ FILENAMES ]

# Kp and Ap index:
Kp, Ap data file: "2001KpAp.txt"

# Solar Activity
Solar wind data file: "ace_merge_25944.lst"
```

There are four sections in this file: global (or NULL) section, `INITIAL CONDITIONS`, `DATE`, and `FILENAMES`.

Another example with references:

```
usr dir: /home/bubukin/usr

[programs]
bin = $(usr dir)/bin
include = $(usr dir)/include
```

The value of `bin` field will be `/home/bubukin/usr/bin` etc.

4 Interface

4.1 Include Instruction

When you work with `parse_conf` library you should include a description file first.

```
#include <parse_conf.h> /* C/C++ code */
```

4.2 Handler Allocation

The library is designed to be thread-safety and you can read/write as many configuration files simultaneously as you would like. The special handler type `pcnf_t` is defined in `parse_conf.h`. You should declare a pointer to this type in your code to work with.

```
pcnf_t* pcnf;
```

Before working with the library you need to get a `pcnf` instance using `pcnf_alloc` function:

```
pcnf_t *pcnf = pcnf_alloc();
```

As usual, the function would return `NULL` if memory allocation would fail by any reason. When the work with the handler is finished, it need to be freed with `pcnf_free`:

```
pcnf_free( pcnf );
```

```
pcnf_t* pcnf_alloc () [Function]
```

Allocates a pointer to the instance of `pcnf_t` type. Returns `NULL` pointer on error.

```
void pcnf_free ( pcnf_t* pcnf ) [Function]
```

Frees the memory allocated by instance of `pcnf_t` type.

4.3 Flags

Flags change the behavior of the library. There are following flags are defined:

```
PCNF_F_DUMP_QUOTED [Flag]
```

If set (unset by default), `pcnf_dump` prints the configuration in quoted format. See [Section 3.4 \[Quoted Format\], page 5](#).

```
PCNF_F_DUMP_EQUAL [Flag]
```

If set (default), `pcnf_dump` uses `=` for fields definitions instead of `:`.

```
PCNF_F_RESOLVE_REFS [Flag]
```

If set (default), functions `pcnf_read` and `pcnf_append` will resolve references in the field definition lines. You can use `pcnf_resolve` function later for references resolution, if this flag is not set.

To set or drop flags the following functions are used:

```
void pcnf_flags_set ( pcnf_t* pcnf, int flags ) [Function]
```

Sets the flags for `pcnf` to value `flags`. For example, to set quoted dump with `=` as a field separator, use

```
pcnf_flags_set( pcnf, PCNF_F_DUMP_QUOTED | PCNF_F_DUMP_EQUAL );
```

```
void pcnf_flags_drop ( pcnf_t* pcnf, int flags ) [Function]
    Drops the flags flags for pcnf. For example, to set unquoted dump, use
        pcnf_flags_drop( pcnf, PCNF_F_DUMP_QUOTED );
```

4.4 Configuration Reading

To read/parse your configuration file you need to use `pcnf_read` or `pcnf_append` functions. They are not different when called first. However, the later one would append an additional information from the second file read to your `pcnf` structure while `pcnf_read` would clean the content before reading the next file. Any of the function will fail if your configuration file contains a syntax error. There is no reading from streams supported at the moment due to the way error messages are reported.

```
int pcnf_read ( pcnf_t* pcnf, char* filename) [Function]
    Reads and parses the file filename. All previous values from the pcnf are removed.
    Function returns PCNF_OK on success.
```

```
int pcnf_append ( pcnf_t* pcnf, char* filename) [Function]
    Reads and parses the file filename. All previous values from the pcnf are saved unless
    they are replaced by same names from filename. As usual, += or := in the fields
    definitions from filename will increment values in pcnf, not replace them. Function
    returns PCNF_OK on success.
```

4.5 Getting Field Values

After you get your file into the `pcnf` structure, you can get field values. All values are stored as stings in the `pcnf` structure. Integers and floats are produced using `atoi` and `atof` functions. There is also a special Boolean type reading which recognizes special words like “true” or “yes” in the configuration file.

```
char* pcnf_sget ( pcnf_t* pcnf, char* gname, char* fname ) [Function]
    Returns the pointer to the field value from section gname with filed name fname. It
    returns NULL if no value found. You are not supposed to change it. Consider this
    as a constant string. To change the meaning of the field, use pcnf_set or pcnf_inc
    functions.
```

```
char* pcnf_sget_dv ( pcnf_t* pcnf, char* gname, char* fname, char* [Function]
                    dv )
    Returns the pointer to the field value from section gname with filed name fname.
    It returns dv (default value) if no field found. You are not supposed to change the
    returned value. Consider this as a constant string. To change the meaning of the
    field, use pcnf_set or pcnf_inc functions.
```

```
int pcnf_iget ( pcnf_t* pcnf, char* gname, char* fname ) [Function]
    Returns the integer value from section gname with filed name fname. Returns 0 if no
    value found.
```

```
int pcnf_iget_dv ( pcnf_t* pcnf, char* gname, char* fname, int dv ) [Function]
    Returns the integer value from section gname with filed name fname. Returns dv
    (default value) if no value found.
```

`double pcnf_fget (pcnf_t* pcnf, char* gname, char* fname)` [Function]
 Returns the double value of the field from section *gname* with filed name *fname*.
 Returns 0 if no value found.

`double pcnf_fget_dv (pcnf_t* pcnf, char* gname, char* fname, double dv)` [Function]
 Returns the double value of the field from section *gname* with filed name *fname*.
 Returns *dv* (the default value) if no field with such a name was found.

`int pcnf_bget (pcnf_t* pcnf, char* gname, char* fname)` [Function]
 Returns 1 if the value of the field from section *gname* with filed name *fname* is interpreted as a “true” statement. Returns 0 if the value is “false” or no field found. For the value to be “true” it needs to be a non-zero number or be the word `yes` or `true` (the case of letters is not important).

`int pcnf_bget_dv (pcnf_t* pcnf, char* gname, char* fname, int dv)` [Function]
 Returns 1 if the value of the field from section *gname* with filed name *fname* is interpreted as a “true” statement. Returns 0 if the value is “false”. It returns *dv* (default value) if no such a field was found. For the value to be “true” it needs to be a non-zero number or be the word `yes` or `true` (the case of letters is not important).

4.6 Getting Arrays

Since version 1.2 there is a simple interface to read arrays directly from the initialization files.

4.6.1 Getting Number of Elements

`size_t pcnf_array_length (pcnf_t* pcnf, char* gname, char* fname, char delimiter)` [Function]
 Returns number of elements in the list from section *gname*, field *field* separated by *delimiter*.

4.6.2 Getting Values

`size_t pcnf_array_iget (pcnf_t* pcnf, int* array, size_t length, char* gname, char* fname, char delimiter)` [Function]
 Reads int values from the list of values from section *gname* with filed name *fname* into array of ints *array*, no more than *length* values. It uses *delimiter* as a separator between the values in the field. It returns number of elements read or 0 for any error.

`size_t pcnf_array_fget (pcnf_t* pcnf, double* array, size_t length, char* gname, char* fname, char delimiter)` [Function]
 Reads double values from the list of values from section *gname* with filed name *fname* into array of doubles *array*, no more than *length* values. It uses *delimiter* as a separator between the values in the field. It returns number of elements read or 0 for any error.

`size_t pcnf_array_fget_dv (pcnf_t* pcnf, double* array, size_t [Function]
length, char* gname, char* fname, char delimiter, double dv)`

Reads double values from the list of values from section *gname* with filed name *fname* into array of doubles *array*, exactly *length* values. If there are less elements than *length* in the field, then it fills other elements with *dv*. It uses *delimiter* as a separator between the values in the field. It returns number of elements read (i.g. *length*) or 0 for any error.

`size_t pcnf_array_iget_dv (pcnf_t* pcnf, int* array, size_t [Function]
length, char* gname, char* fname, char delimiter, int dv)`

Reads int values from the list of values from section *gname* with filed name *fname* into array of ints *array*, exactly *length* values. If there are less elements than *length* in the field, then it fills other elements with *dv*. It uses *delimiter* as a separator between the values in the field. It returns number of elements read (i.g. *length*) or 0 for any error.

4.6.3 Array Usage Example

Suppose there is an ini file named “testdata.ini”.

```
[lists]
flist = 0, 1, 0.2,3.14 # total 4 values
```

The following program reads these values into array and outputs them.

```
#include <stdio.h>
#include <parse_conf.h>

int main()
{
    size_t len;
    size_t i;
    double *A;
    pcnf_t *pcnf = pcnf_alloc();

    pcnf_read( pcnf, "testdata.ini" );

    /* reading flist */
    len = pcnf_array_length( pcnf, "lists", "flist", ',' );

    A = malloc( len*sizeof( double ) );
    len = pcnf_array_fget( pcnf, A, len, "lists", "flist", ',' );
    for ( i = 0; i < len; ++i ) printf( "%lf\n", A[i] );

    free( A );
    pcnf_free( pcnf );

    return 0;
}
```

4.7 Getting Lists of Values

4.7.1 List Types

You also (since version 1.1) can specify a list of values in one field. This can be either strings, integers, or floats. Only one fixed type per list. There are 3 types predefined to operate with lists:

`pcnf_slist` [Type]

This is the type for storing the list of strings.

`pcnf_ilst` [Type]

This is the type for storing the list of integers.

`pcnf_flist` [Type]

This is the type for storing the list of double precision values.

4.7.2 Allocation Lists

Before getting any of the values, you should declare the pointer to the list you wish to use. For example

```
pcnf_slist *pslist;
```

to declare the pointer to the list of strings. Next step would be to allocate the memory for the list. The following functions perform necessary allocation. They return a pointer to the allocated memory or `NULL` if allocation was not successful:

`pcnf_slist * pcnf_slist_alloc ()` [Function]

`pcnf_ilst * pcnf_ilst_alloc ()` [Function]

`pcnf_flist * pcnf_flist_alloc ()` [Function]

The following function have to be called to release the memory:

`void pcnf_slist_free (pcnf_slist * ps)` [Function]

`void pcnf_ilst_free (pcnf_ilst * ps)` [Function]

`void pcnf_flist_free (pcnf_flist * ps)` [Function]

4.7.3 Getting Variables

Every element from the read list can be accessed one by one in the order they were listed. Any time, the list can be reset and reading can start from the beginning. The length of the list is available at any time.

The following functions are used to get the list of values:

`int pcnf_slist_get (pcnf_t * pcnf, pcnf_slist * pslist, char * group, char * field, char delim)` [Function]

Returns 1 if the option exists and 0 if there were no such option in the ini file. It reads the list into `pslist` from the `$(group)(field)` using the delimiter specified in `delim`.

`int pcnf_ilst_get (pcnf_t * pcnf, pcnf_ilst * pilist, char * group, char * field, char delim)` [Function]

Returns 1 if the option exists and 0 if there were no such option in the ini file. It reads the list into `pslist` from the `$(group)(field)` using the delimiter specified in `delim`.

```
int pconf_flist_get ( pconf_t *pconf, pconf_slist *pflist, char *      [Function]
                    group, char *field, char delim )
```

Returns 1 if the option exists and 0 if there were no such option in the ini file. It reads the list into *pflist* from the $group(field)$ using the delimiter specified in *delim*.

The following functions are used to receive the next available value from the list. All following functions move pointer to the next unread value.

```
char* pconf_slist_next ( pconf_slist *ps )                          [Function]
```

Returns the next unread string from the list of strings. It returns NULL in the end of the list was reached.

```
int pconf_ilst_next ( pconf_ilst *ps, int *pa )                    [Function]
```

Reads the value of the next unread integer from the list of integers and sets it to the **pa*. It returns 1 on success and 0 if the end of the list was reached.

```
int pconf_flist_next ( pconf_ilst *ps, double *pa )               [Function]
```

Reads the value of the next unread double from the list of doubles and sets it to the **pa*. It returns 1 on success and 0 if the end of the list was reached.

The following functions reset the list such that all elements are marked unread.

```
void pconf_slist_reset ( pconf_slist *ps )                         [Function]
```

```
void pconf_ilst_reset ( pconf_ilst *ps )                           [Function]
```

```
void pconf_flist_reset ( pconf_flist *ps )                         [Function]
```

The following functions return the length of the list:

```
size_t pconf_slist_length ( pconf_slist *ps )                     [Function]
```

```
size_t pconf_ilst_length ( pconf_ilst *ps )                       [Function]
```

```
size_t pconf_flist_length ( pconf_flist *ps )                     [Function]
```

4.7.4 Usage Example

The following program reads the list of integers from the file and outputs it to the standard output.

The data (file “example.ini”)

```
list example = 1, 2, 3, 4, 5, 6, 7
```

The program:

```
#include <stdio.h>
#include <assert.h>
#include <parse_conf.h>
```

```
int main()
{
    pconf_t *pconf;
    pconf_ilst *pilst;
    int i;
    int flag;
```

```

pcnf = pcnf_alloc();
pcnf_read( pcnf, "example.ini" );

/* get the list */
pulist = pcnf_elist_alloc( );
assert( pulist );
flag = pcnf_elist_get( pcnf, pulist, NULL, "list example", ',' );
assert( flag );

/* print the list */
do {
    flag = pcnf_elist_next( pulist, &i );
    if ( flag ) printf( "%d\n", i );
}
while( flag );

pcnf_elist_free( pulist );
pcnf_free( pcnf );
return 0;
}

```

4.8 Setting Field Values

You can set different values if desired. It can be useful for generation configuration files. It also can be useful to replace some values in other configuration files or to resolve some undefined variables if you use your configuration files as templates.

`int pcnf_set (pcnf_t* pcnf, char* gname, char* fname, char* value)` [Function]
Sets the string value of field *fname* in section *gname* to *value*. Function actually duplicates the string *value*.

`int pcnf_inc (pcnf_t* pcnf, char* gname, char* fname, char* value)` [Function]
Appends the string *value* to the value of the field *fname* in section *gname*.

4.9 Printing Configuration

`void pcnf_dump (pcnf_t* pcnf, FILE* fp)` [Function]
Dumps the configuration into the stream *fp*. To print the configuration in a standard parse-conf format to the screen, use `pcnf_dump(pcnf, stdout);`

which can be useful even for test what variables were read.

4.10 Cleaning Configuration

`void pcnf_clean (pcnf_t* pcnf)` [Function]
Removes all content from *PCNF*. It does not free the memory though.

4.11 Resolving References

`int pcnf_resolve (pcnf_t* pcnf)` [Function]
 Resolves all references. This function is useful when flag `PCNF_F_RESOLVE_REFS` is not set. You can combine a few configuration files together and resolve cross references in them. Another application would be to provide a reference to the file with undefined references in it. For example, you can decide to use this if reference values can be known only during an execution time of your program.

4.12 Data Direct Access

The following functions provide direct access to the configuration data. They can be used to list all available sections or fields in the configuration file. They also provide a different way to access field values if necessary. However, it is advised to use higher level functions from `pcnf_get` class for getting the data.

There are two additional types defined:

`pcnf_sec_t` [Type]
 A section type. It is a pointer to a structure that contains all information about a particular section. This type is used by Data Direct Access functions.

`pcnf_field_t` [Type]
 A field type. It is a pointer to a structure that contains all information about a particular field. This type is used by Data Direct Access functions.

`size_t pcnf_sec_start (pcnf_t* pcnf, pcnf_sec_t* psec)` [Function]
 Initialize section *psec* by the first section (always a NULL section). Returns total number of sections in the configuration including NULL section. *psec* should be a pointer to the previously declared `pcnf_sec_t`. This function must be called before any other `pcnf_sec` functions.

`pcnf_sec_t pcnf_sec_next (pcnf_sec_t sec)` [Function]
 Getting some section *sec*, the function returns the next section or NULL if *sec* is the last section. It is used to list all section one by one.

`char* pcnf_sec_name (pcnf_sec_t sec)` [Function]
 Returns the name of section *sec*. This name should not be changed or alter in the program.

`pcnf_sec_t pcnf_sec_find (pcnf_t* pcnf, char* name)` [Function]
 Returns the section from *pcnf* with the name *name*

`size_t pcnf_field_start (pcnf_sec_t sec, pcnf_field_t* pfield)` [Function]
 Initialize field type pointer *pfield* by the first field from section *sec* or set *pfield* to NULL if the section is empty. It returns number of fields in *sec*.

`pcnf_field_t pcnf_field_next (pcnf_field_t field)` [Function]
 Returns the next field after *field* or NULL if *field* is the last field in the section.

`char* pcnf_field_name (pcnf_field_t field)` [Function]
 Returns the name of the field *field*. This value should not be changed.

`char* pcnf_field_value (pcnf_field_t field)` [Function]
Returns the string value of the field *field*. This value should not be changed.

`pcnf_field_t pcnf_field_find (pcnf_sec_t sec, char* name)` [Function]
Returns the field from section *sec* with the name *name* or NULL if nothing found.

4.13 Section Listing

The example below shows how to use direct access functions to print all available sections from `testdata.ini` file except the NULL-section (which has no name by the way):

```
pcnf_t *pcnf = pcnf_alloc();
pcnf_sec_t sec;

pcnf_read( pcnf, "testdata.ini" );
(void) pcnf_sec_start( pcnf, &sec );
sec = pcnf_sec_next( sec ); /* skip the NULL section */
while( sec )
{
    printf( "Section: %s\n", pcnf_sec_name( sec ) );
    sec = pcnf_sec_next( sec );
}

pcnf_free( pcnf );
return 0;
```

5 Compiling and Linking

After you properly installed the library, you can include the header in your program by

```
#include <parse_conf.h>
```

and compile it. To link your code against the library, you need `-lparse_conf` flag for your linker. For example,

```
gcc <you_program_files> -lparse_conf -o program_name
```

6 pcnfdump Utility

`pcnfdump` is a utility to read the configuration file and dump its content resolving references if necessary. It reads a configuration file specified and outputs what it have read in some of the standard format that `parse_conf` library can read. It is useful to check the substitutions of references made. It also can be used to generate simpler configuration files without references from more general `parse_conf` library format.

```
Usage: pcnfdump [-o output_file] [-xQnhV] input_file
  -x      variable definitions with : instead of =
  -Q      variable definitions should be quoted
  -n      do not resolve references in definition lines

  -V      version and other information about the program
  -h      this usage message
```

Appendix A Index

\$

\$ 4

%

% 4

*

* 12

\

\" 3

\" 5

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

\" 3

C

cleaning configuration 14

comment 4

compiling the library 17

ConfigParser in Python 4

configuration file 3

configuration format 3

D

definition line 3

dumping 14

F

field 3

field name 4

field type 15

flags 8

format 3

G

getting fields 9

global reference 4

global section 3

group 3

H

handler 8

I

initialization file 3

installation 2

instance 8

Interface 8

L

linking with the library 17

list 5, 10, 12

local reference 4

N

NULL section 3

P

pcnf_alloc 8

pcnf_append 9

pcnf_array_fget 10

pcnf_array_fget_dv 11

pcnf_array_iget 10

pcnf_array_iget_dv 11

pcnf_array_length 10

pcnf_bget 10

pcnf_bget_dv 10

pcnf_clean 14

pcnf_dump 14

PCNF_F_DUMP_EQUAL 8

PCNF_F_DUMP_QUOTED 8

PCNF_F_RESOLVE_REFS 8

pcnf_fget 10

pcnf_fget_dv 10

pcnf_field_find 16

pcnf_field_name 15

pcnf_field_next 15

pcnf_field_start 15

pcnf_field_t 15

pcnf_field_value 16

pcnf_flags_drop 9

pcnf_flags_set 8

pcnf_flist 12

pcnf_flist_free 12

pcnf_flist_get 13

pcnf_flist_length 13

pcnf_flist_next 13

pcnf_flist_reset 13

pcnf_free 8

pcnf_iget 9

pcnf_iget_dv 9

pcnf_ilst 12

pcnf_ilst_free 12

pcnf_ilst_get 12

pcnf_elist_length	13
pcnf_elist_next	13
pcnf_elist_reset	13
pcnf_inc	14
pcnf_read	9
pcnf_resolve	15
pcnf_sec_find	15
pcnf_sec_name	15
pcnf_sec_next	15
pcnf_sec_start	15
pcnf_sec_t	15
pcnf_set	14
pcnf_sget	9
pcnf_sget_dv	9
pcnf_slist	12
pcnf_slist_free	12
pcnf_slist_get	12
pcnf_slist_length	13
pcnf_slist_next	13
pcnf_slist_reset	13
pcnf_t	8
pcnfdump	18
printing configuration	14

Q

quote sign	3
quoted format	5

R

read file	9
reference	4

S

section	3
section declaration	3
section name	3
section type	15
separator	5
setting fields	14
space sign	3
special sequences	3
splitting long lines	3
substitution rule	5

T

tabulation sign	3
-----------------	---

V

variable	3
variable definition	3
variable name	4
variable resolution	4