GetPot Version 1.1.16

Powerful Input File and Command Line Parser

Frank R. Schaefer
Email: fschaef@users.sourceforge.net

March 1, 2007

Contents

1 Installation **3**

2 Overview **3**
2.1 Other Constructors 4

3 Options **4**
3.1 Arguments that follow arguments 5
3.2 Nominus Followers 6
3.3 Directly followed options 7
3.4 Tails 7

4 Variables **8**

5 Flags **9**

6 Nominus Arguments **9**

7 Direct access to command line arguments **10**

8 Input files **10**

9 Prefixes **11**

10 UFOs - Unidentified Flying Objects **12**
10.1 Unidentified Arguments 12
10.2 Unidentified Options 13
10.3 Unidentified Flags 13
10.4 Unidentified Variables 13
10.5 Unidentified Sections 14
10.6 Unidentified Nominuses 14

11 Dollar Bracket Expressions **14**
11.1 String Operations 14
11.2 Arithmetic Operations 17
11.3 Comparison Operators 17
11.4 Conditional Expansion 18

11.5 Vector and String Subscription 18
11.6 Macro Calls 18

Abstract

The two basic methods to pass parameters to the `main()`-routine are: *input files* and *command line arguments*. For small scale programs these input methods allow to change parameters without having to recompile or having to create an input file parser. Even for large programs input files and command line arguments are a comfortable way to replace annoying graphical user interfaces. When debugging complicated code, it is essential to be able to isolate code fragments into a lonely `main()`-routine. In order to feed these isolated code fragments with realistic data, sophisticated command line and input file parsing becomes indispensable.

`GetPot` allows to parse input files and the command line in a very efficient manner. `GetPot` itself is a small piece of code, completely contained in a header file. This way the installation is very easy and platform independent. The present article discusses the C++ implementation of `GetPot`. However, `GetPot` has been ported to Python, Java and a Ruby. Also, the code has been organized in a way that facilitates porting GetPot to other languages.

Traditionally, programmers relied on the *getopt* library [**?**] to handle basic command line interpretation. In recognition of the existing getopt library it is brand with an anagram: `GetPot`. The software as well as this document is distributed under GNU Lesser General Public License[1] terms (LPGL).

Introduction

a prova di manomissione

In order to implement a tamperproof command line parsing, one usually has to write a significant amount of code. Using `GetPot` allows to shrink code fragments such as

```
...
   int      i = 0;
   double   v = 0;
   char*    endptr = 0;
...
   if( i < argc ) {
       v = strtod(argv[i], &endptr);
       if( endptr == argv[i] ) // argv[i] was not a number
          v = 9.81;            // set default value
   }
...
```

to a single line

```
    const double V = cl.get(i, 9.81);
```

In fact, `GetPot` allows you to do much more sophisticated things in not more than one single program line. Additionally, `GetPot` provides a parser to handle input files such as 'example.pot':

```
...
   webpage   = http://getpot.sourceforge.net/GetPot.html
```

[1] This means basically, it is for free but it can still be used in commercial products. You should have received a copy of the LPGL along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

```
   user      = 'G. V. V. Bouche'      # whitespace requires quotes
   clicks    = 231    # [1/s]
   factor_x  = 1.231  # [m/s^2]

   [vehicle]
   wheel-base   = 2.65                # [m]
   initial-xyz  = '100.  0.1  5.0'    # [m]
...
```

and read out its contents in the same way the command line was parsed. There are a few special things to know. In some aspects parsing input files differs from parsing the command line.

1 Installation

`GetPot` is a project hosted at Sourceforge. There is a web page that provides download-able `GetPot` libraries for different programming languages:

`http://getpot.sourceforge.net/`

To install `GetPot`, simply copy the file `GetPot` somewhere into the file system where your compiler can find it (`/usr/include` or so[2]). As a requirement you should have installed the STL-Library which usually comes with any C++ compiler distribution.

The easiest way to get started is to go through the example files and copy/paste the things you need into your own program. This way you learn `GetPot` in less than 10 minutes. The following text is mainly written, because every library should have a manual. It basically describes what you would understand anyway when you go through the examples.

2 Overview

reazione, riflesso

When an experienced programmer starts to write the `main()`-routine in C or C++ he produces by reflex a certain code fragment. Programmers who tasted `GetOpt` write by reflex an extended version such as the following:

```
#include<iostream>
#include<GetPot>
...
int main(int argc, char** argv)
{
   GetPot    cl(argc, argv);
...
```

The first object created in the application is of type `GetPot`. It uses `argc` and `argv` to build its internal database of command line arguments. This is all that is required to start doing fancy things. If one wants to parse an input file, one has to specify the filename to the constructor, i.e.

```
...
   GetPot    ifl("sprites.pot");
...
```

[2] If you have no root access on your machine, copy it somewhere in your directory tree and specify `-I/home/genius/my_include/` when using g++ - supposed that you copied `GetPot` into `/home/genius/my_include/`.

defines the file 'sprites.pot' to be the input file for GetPot's internal database. All functions explained in the following text work independently of the way the database was constructed. They can be applied to any object of type GetPot, whether it was build from a command line or an input file. Input files, however, provide some special features described in section 8. Distributions of GetPot versions 1.0 and above contain a *getpot-mode* for emacs to highlight GetPot input files.

There are four different types of command line arguments that are known to GetPot:

Options: Arguments like --help, -h, and --force can be easily checked for existence. Further, an elegant means is provided to parse arguments that follow specific arguments.

Variables: Variables can be defined on the command line and read out as numbers, strings or mixed type vectors.

Flags: GetPot checks if an argument, or any option (argument starting with a single '-') contains a specific letter.

Nominus Arguments: GetPot lets you iterate through arguments that do not start with a minus-sign.

The very basic idea of the GetPot is to specify the expected return type through a default argument[3]. There are three basic types known to GetPot: int, double, and std::string. A set of functions of the same name with an overloaded version for each type is called a *function group*.

2.1 Other Constructors

As mentioned in the previous paragraphs, the command line or configuration files are parsed by a call to the constructor of GetPot. There are the two obvious constructors:

- GetPot(arg, argv): initiates the parsing of the command line.
- GetPot(Filename): initiates the parsing of a configuration file.

To provide the user with more flexibility, GetPot provides two more constructors. They determine the way strings are split up into vectors and what strings determine the start and the end of a comment:

- GetPot(arg, argv, FieldSeparator=','): initiates the parsing of the command line. Arguments are split into vectors using FieldSeparator (see section 4). The default value is ','.
- GetPot(Filename, CommentStart=''#'', CommentEnd=''\n'', FieldSeparator='',''): initiates the parsing of a configuration file. Anything inside CommentStart and CommentEnd is considered to be a comment. Arguments are split into vectors using FieldSeparator.

3 Options

The easiest way to check if an argument is specified on the command line is to use the search()-function. It returns true in case the option is found and false in case it was not. In addition to that, GetPot provides a search() function with a variable argument list allowing to check elegantly for multiple options that are equivalent. Example:

[3]Originally, C++ does not allow a function overloading with respect to the return type. The default argument must have the same type as the variable it will be assigned to. Therefore, functions are overloaded with respect to the default argument, which at the same time defines the return type.

```
...
     bool  be_nice_f = cl.search("--nice");
     if( cl.search("--do-nothing") ) exit(0);
     if( cl.search(4, "--help", "-h", "--hilfe", "--sos") ) {
        // print some information about how the program works
     }
...
```

The '4' as a first argument to `search()` indicates that four strings follow that represent equivalent options. `search()` functions belong to a class of *cursor related functions*. Cursor related functions are a very convenient means to parse the command line. In order to understand how they work, one has to understand how command line arguments are lined up. A command line given as

```
> hello aux --recompile ... my_input.txt
```

is stored in an array as shown in **??**. The the `search()` functions set the cursor to a certain position in the array. Others increase the cursor position. The two member functions

```
void        disable_loop();
void        enable_loop();
```

allow to specify if the cursor is allowed to go back to the beginning, if no match is found until the end of the array. The default behavior is 'yes'. In case one needs to allow multiple occurrence of an option (e.g. as `-I` in gcc), the wrapping has to be turned off to avoid parsing the same option twice. Before parsing these kinds of options, one has to reset the cursor position by the function:

```
void        reset_cursor();
```

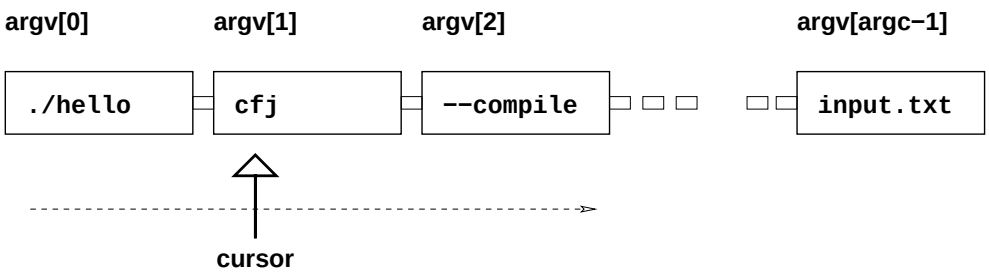

Figure 1: Command line arguments lined up in the `argv`-array. A cursor iterates over the elements.

3.1 Arguments that follow arguments

When an argument is found by `search()`, the cursor is set to its position. Now, the `next()` functions allow to parse through terms that follow. A line like

```
> myprog --size 14.2341 -i fichier.txt
```

can be parsed using `search()` and `next()` as follows:

```
...
     if( ! cl.search("--size") ) return;
     const double XSize = cl.next(0.);

     if( ! cl.search(2, "-i", "--file") ) return;
     const std::string infile = cl.next("dummy.txt");
...
```

Note that the desired return type is specified by the default argument. `XSize` is of type `double` so the default argument for `next()` is of type `double`. If the argument following '`--size`' cannot be converted to a `double`, then the default value '0.' will be assigned to `XSize`. Respectively the same thing happens with `infile` where '`dummy.txt`' specifies that a `std::string` parameter is required.

The above code may still seem a bit cumbersome. That is why `GetPot` provides the `follow()` functions. These functions combine the search for an argument with the search for a following argument. The above code can be rewritten as:

```
...
    const double      XSize  = cl.follow(0., "--xsize");
    const std::string infile = cl.follow("dummy.txt", 2, "-i","--file");
...
```

The `search()` and `follow()` functions set the cursor to the next argument that matches, starting from the current cursor position. These functions can also be used to parse arguments that occur multiple times. The following code shows how multiple occurrence of an argument can be parsed.

```
...
  vector<double> accelerations;                 // [m/s^2] accelerations
  const double   v_d = cl.follow("-a", 9.81); // max.  9.81 [m/s^2]
  // read all arguments that start with '-I'
  cl.init_multiple_occurrence();
  while(fabs(v_d) <= 9.81)
    accelerations.push_back(cl.follow("-a", 9.81));

  // re-enable wrapping around for following parsing operations ...
  cl.enable_loop();
...
```

Since every time when one needs to parse options with multiple occurrence, one needs to call `disable_loop()` and `reset_cursor()`, these two functions are combined into one, for convenience:

```
void          init_multiple_occurrence();
```

The above code parses command lines like:

```
> myprog ... -a 3.1 ... -a -8.2 ... -a 0.2 ... -a 1.02 ...
```

3.2 Nominus Followers

Two functions support the listing of files or similar command line arguments that start not with a minus, but follow an option:

```
const vector<string> nominus_followers(const char* Option);
const vector<string> nominus_followers(const unsigned No, ...);
```

Both functions return a list of strings that follow an option. The second function allows the specification of multiple options, the number of passed options has to be passed as the first argument. The following code fragment

```
const vector<string> inputs  = cl.nominus_followers("-i");
const vector<string> outputs = cl.nominus_followers(2, "-o", "--output",
                                                    "--ausgabe");
```

when fed with the following command line:

```
> application -i main.cpp example.cpp audio.cpp -o video.out --output
  libAudio.so libKeyBoard.so libVehicle.so --ausgabe Compressed RGB
```

will produce two vectors `inputs` and `outputs` where the first contains the strings 'main.cpp'. 'example.cpp', and 'audio.cpp', The second vector will contain 'video.out', 'libAudio.so', 'libKeyBoard.so', 'libVehicle.so', 'Compressed', and 'RGB'.

3.3 Directly followed options

In the previous section, it is discussed how `GetPot` handles arguments that follow other arguments on the command line. Popular programs, such as 'gcc' parse information that directly follows a string without any whitespace, such as '`-I/usr/local/include/GL`'. In this case '`-I`' indicates that a include path is to be set and the following character chain represents the path to be used. Of course, `GetPot` provides an easy means to access these strings: the `direct_follow()`-functions. This is described in the present section. However, take a look at the *tail()*- functions, section 3.4. They are even easier and solve most probably all your demans. For now, here is an example for using the direct follow functions:

```
...
    vector<string>     incl_path;
    const std::string  path = cl.direct_follow("", "-I");
    // read all arguments that start with '-I'
    cl.init_multiple_occurrence();
    while(path != "" ) {
        incl_path.push_back(string(path));
        path = cl.direct_follow("", "-I");
    }
...
```

3.4 Tails

Options that are appended to a starting string such as '-I' or '-L' in a C compiler, can be read very easily using the tails functions, of which there are three:

- `string_tails(StartString)`: returning a vector of *strings* that are appended to any option starting with `StartString`.
- `int_tails(StartString, Default = 1)`: returning a vector of *integers* that are appended to any option starting with `StartString`.
- `double_tails(StartString, Default=-1.0)`: returning a vector of *double precision float numbers* that are appended to any option starting with `StartString`.

The latter two functions require a conversion of a string to a number. This can potentially fail. So, you can provide a default value by which you determine that an error occured. If a value in the returned list is equal to that default value, you assume that the user was not able to specify a decent number. By default, the default value is minus one.

4 Variables

Variables are a very elegant feature to pass arguments to your program. In the style of 'awk' one can specify some variables, assign values to them, and read them from inside your program. A 'variable' is defined on the command line like

variable-name '=' value.

Note that no blanks are allowed. Quotes can be used, though, to pass longer strings or vectors. Variables are accessed through the function call operator. In the same way as the `next()` and `follow()` function groups, the function call operator determines its return type through the default argument. By a code fragment like

```
...
  const double       initial_velocity = cl("v0", 10.);              // [m/s]
  const std::string  integration_type = cl("integration", "euler");
  const int          no_samples_per_sec = cl("samples", 100);       // [1/s]
...
```

one can parse a command line like:

```
> myprog ... v0=14.56 integration=runge-kutta samples=500 ...
```

`GetPot`, can do even more. It is able to treat input vectors, of mixed type. This allows to have command line arguments like

```
> myprog ... sample-interval='-10. 10. 400' \
             color-mode='32 RGB 0.4 0.2 0.5' ...
```

Vectors variables are also read with the function call operator. However, the position in the vector has to be specified by an index. The total number of elements in a vector can be determined through the function `vector_variable_size(VariableName)`. Here is an example of how to parse the above command line:

```
...
 if( cl.vector_variable_size("sample-interval") < 1 ) {
    cerr << "error: sample interval, argument does not fit format:\n";
    cerr << "       MINIMUM MAXIMUM NO_SAMPLES\n".
    cerr << "       need at least the minimum !\n";
 }
 const double Min = cl("sample-interval", 0., 0);          // [s] default 0
 const double Max = cl("sample-interval", 10., 1);         // [s] default 10
 const int    NoSamples = cl("sample-interval", 200, 2); // sample no.
...
 const int          BPP  = cl("color-mode", 64, 0);
 const std::string  Mode = cl("color-mode", "Grayscale", 1);
 if( ! strcmp(Mode,"RGB") && BPP < 24 )
    cerr << "RGB not possible with less than 24 bits per pixel.\n";
...
```

The third argument to the ()-operator specifies the position in the vector. The first element of the vector 'sample-interval' (index 0) defines the minimum Min. The second argument the maximum (index 1) and the third element (index 2) the number of samples. Respectively the 'color-mode' information is parsed.

5 Flags

Flags are activated by single letters inside an argument or option[4]. GetPot has two functions to access flags:

options_contain(const char* Flags): Checks if any option (argument that does start with a single '-') contains one of the flags specified in the string Flags.

argument_contains(unsigned Idx, const char* Flags): Checks if argument number Idx contains one of the flags specified in Flags.

Here is an example:

```
...
    const bool  verbose_f = cl.options_contain("vV");
    const bool  extract_f = cl.argument_contains(1, "xX");
    const bool  create_f  = cl.argument_contains(1, "cC");
    if( create_f && extract_f )
       { cerr << "cannot create and extract at the same time.\n"; exit(-1); }
...
```

At first, it is checked if any option (argument that starts with a minus) contains a letter 'v' or 'V'. If it does, then the verbose flag will be set. The letters 'x', 'X', 'c', and 'C' in the first argument indicate if one uses the program to extract or create a new file. Correspondingly the flags are set.

6 Nominus Arguments

In order to quickly go through the arguments that do not start with a minus, the next_nominus() function is provided. Each call to this function returns the following argument that does not have a minus. If there is no further argument, this function return '0'. Example:

```
...
     vector<string>      files;
     const std::string   nm = cl.next_nominus();
     while( nm != 0 ) {
          file.push_back(nm);
          nm = cl.next_nominus();
     }
...
```

Here the nominus arguments are read one after the other into an array of strings containing some filenames. If this is all one needs, one is better of asking directly for a vector of all nominus arguments:

```
    vector<string>    files = cl.nominus_vector();
```

This line does the same as the slightly more complicated piece of code above.

[4] A famous program using these type of options is 'tar'. With strings like 'xzvf' this program allows to state very concisely a desired behavior (extract 'x', unzip 'z', verbose 'v' the following file 'f')

7 Direct access to command line arguments

The very simplest way to access the argument list one is to use the `[]`-operator:

```
while( cl[5] ) cout << cl[5] << endl;
```

The `[]`-operator returns a zero pointer in case that one tried to reference an option that does not exists. Similar to that, the `get()`-functions allow to specify a type through a default argument, so that one writes for example:

```
const double Acceleration = cl.get(5, 9.81);
```

to use argument number five as the value of a certain `double` variable. The high flexibility of `GetPot`, however, makes these methods seem to be a little outdated. Fixing the position of an argument in a argument list, is a restriction that makes the use of a program unnecessarily difficult. As seen preceeding sections, one can do much better without adding any more programming effort.

8 Input files

Input files are parsed a little differently from the command line. Perhaps the most important feature are the '#'-comments that allow to add text into the file without being actually parsed:

```
# -*- getpot -*-  activate emacs 'getpot-mode'
# FILE:     "example.pot"
# PURPOSE: some examples how to use GetPot for input file parsing.
#
# (C) 2001 Frank R. Schaefer
#
# License Terms: GNU Lesser GPL, ABSOLUTELY NO WARRANTY
###############################################################################
v0 = 34.2     # [m/s^2] initial speed of point mass
```

As can be seen in the same example, the variable assignments now allow whitespace between a left hand value `v0` the assignment operator '`=`' and a right hand value `34.2`.

One thing is crucially different in input files: sections. In input files, the basic advantage of sections is to reduce the variable length. Imagine, you have a parameter `weight` occurring multiple times: for the total vehicle, for each tire, for a load in the trunk etc. Now in order give each one a unique identifier one would have to call them `total_vehicle_weight`, `vehicle_tires_front_left_weight` and so on. Here is where sections become handy:

```
...
    [vehicle]
    length   = 2.65                # [m]
    initial-xyz = '100.  0.1  5.0' # [m]

    # coefficients of magic formula [Pajeka, et al.]
    [./tires/front/right]
    B = 3.7976     C = 1.2     E = -0.5     D = 64322.404
    [./tires/front/left]
    B = 3.7976     C = 1.2     E = -0.5     D = 64322.404
    [./tires/rear/right]
    B = 3.546     C = 1.135    E = -0.4     D = 59322.32
    [./tires/rear/left]
```

```
    B = 3.546     C = 1.11     E = -0.32    D = 59322.32


    [../chassis]  # i.e. vehicle/chassis
    Roh = 1.21;   # [kg/m^3] density of air
    S   = 5.14;   # [m^2]    reference surface
    Cd  = 0.45;   # [1]      air drag coefficient
...
```

In section 4 it was explained how to read out variables. The same functions work, of course, for a database build up on a parsed file. The only difference is that sections produce suffixes in front of variables and options. A database fed with a file as the above one, can be queried as follows:

```
...
    front_tire.B = ifile("vehicle/tires/front/right/B", 0.);
    front_tire.C = ifile("vehicle/tires/front/right/C", 0.);
    front_tire.D = ifile("vehicle/tires/front/right/D", 0.);
    front_tire.E = ifile("vehicle/tires/front/right/E", 0.);
...
```

Note that there are two special indicators in a section label:

`./` Take the actual section and append the following name to it.

`../` Go back to the parent section and create a name appended to the name of the parent.

Finally, an empty label like `[]`, resets the section suffix to nothing. Speaking in name space terminology, we are back to the `global` name space.

9 Prefixes

When reading out variables, options and flags in a large section tree, it can be very messy if one has to write the full variable name such as

```
"vehicle/tires/front/right/B"
```

Here, the prefix feature comes convinient. It allows to narrow the search space for arguments, variables, options and flags. Only such objects are considered that start with the given prefix. The prefix itself is then cut of the object before investigation. In specific this prefix can be a section name, so that a code fragment like

```
...
    front_tire.B = ifile("vehicle/tires/front/right/B", 0.);
    front_tire.C = ifile("vehicle/tires/front/right/C", 0.);
    front_tire.D = ifile("vehicle/tires/front/right/D", 0.);
    front_tire.E = ifile("vehicle/tires/front/right/E", 0.);
...
```

can be rewritten as

```
...
    ifile.set_prefix("vehicle/tires/front/right/")
    front_tire.B = ifile("B", 0.);
    front_tire.C = ifile("C", 0.);
```

```
    front_tire.D = ifile("D", 0.);
    front_tire.E = ifile("E", 0.);
...
```

which is, of course, much more readable. However, one should not forget to set or reset the prefix to the correspondent section (such as `""` for the root section).

10 UFOs - Unidentified Flying Objects

Some users tend to pass wrong options to a program on the command line, to define sections in configuration files that nobody cares about (e.g. 'vehicle/front-tire-groop') and tiny input errors can prevent a program from functioning. In order to allow the program to tell the jittery user that he did something that nobody understands, GetPot provides UFO detection !

Indeed, these unidentified command line arguments that fly around without ever being processed, these variables in configuration files that do not serve any purpose, command line switches that do not switch anything - these things are easily detected and the programmer can decide what he wants to do about it.

> **NOTE:** UFO detection is prefix dependent ! In case you were using prefixes, you must reset the prefix to the section you are investigating[5].

The following sections discuss functions where the valid options are explicitly passed to the function. However, since version 1.1.5, GetPot traces the access to command line arguments. If it is enough evidence that no function ever read a command line argument, then you might use simply the following functions:

1. `unidentified_arguments()`
2. `unidentified_options()`
3. `unidentified_variables()`
4. `unidentified_sections()`
5. `unidentified_nominuses()`

Each of them returns a vector of strings, i.e. the arguments, options, variables, sections or nominuses that are 'untouched.' Call these functions at the end of your command line/configuration file treatment.

10.1 Unidentified Arguments

In order to detect unidentified flying arguments one can uses the following functions:

```
vector<string>
unidentified_arguments(unsigned Number,
                       const char* Known, ...) const;

vector<string>
unidentified_arguments(const vector<string>& Knowns) const;
```

[5] That means to `""` for example, in case you want to refer to the root section

In the first case, `Number` specifies the number of known arguments and the following list of `const char*`-pointers specify the list of known arguments. In the second case, the known arguments are collected in a string vector. This is particularly useful, in case the known arguments are dynamic (nominus arguments as filenames, etc.).

Any argument on the command line, that is not contained in the specified list of known arguments is listed in the string vector that is returned by these functions.

10.2 Unidentified Options

Similarly, unidentified flying options can be detected by the functions:

```
vector<string>
unidentified_options(unsigned Number,
                     const char* Known, ...) const;

vector<string>
unidentified_options(const vector<string>& Knowns) const;
```

These functions work the same way as the functions for unidentified flying arguments, but only argument that do start with at least one '-' are considered.

10.3 Unidentified Flags

Flags, i.e. letters in arguments/options that activate or deactivate certain switches can be checked using the function:

```
string
unidentified_flags(const char* Known,
                   int ArgumentNumber /* =-1 */) const;
```

This function operates in two modes. In first mode, if the second argument is omitted or set to -1, then all options starting with a single '-' are considered and checked if there is any letter in them that is not in *Known*. *Known* is simply a string concatenating all possible flags (such as 'xcvfjt' for tar). In the second mode, the argument number `ArgumentNumber` on the command line is checked for flags (argument 1 for example in tar). The flags that are not contained in `Known` are listed in the returned string.

10.4 Unidentified Variables

Unidentified flying variables on the command line or in configuration files may be confusing the same way as arguments, options and flags. Therefore the functions:

```
vector<string>
unidentified_variables(unsigned Number,
                       const char* Known, ...) const;

vector<string>
unidentified_variables(const vector<string>& Knowns) const;
```

provide UFO detection for variables in the usual manner as the described above.

10.5 Unidentified Sections

Users may even define nonsense sections or mess around with the *../*- and `./`-parts in the section labels. The result is a unreadable configuration file. To detect such mischievous settings the functions

```
vector<string>
unidentified_sections(unsigned Number,
                      const char* Known, ...) const;
vector<string>
unidentified_sections(const vector<string>& Knowns) const;
```

detect unidentified flying sections.

10.6 Unidentified Nominuses

The last category of UFOs are nominus arguments that are not recognized as filenames, variables or section names. They are detected by the functions

```
vector<string>
unidentified_nominuses(unsigned Number,
                       const char* Known, ...) const;

vector<string>
unidentified_nominuses(const vector<string>& Knowns) const;
```

following the usual UFO detection procedure.

11 Dollar Bracket Expressions

Since version 1.0 the GetPot parser provides a sophisticated function to handle several arithmetic and string operations. In some cases this can significantly facilitate the writing of a configuration file. The so called dollar bracket expressions constitute a very simple *lisp-like* language. Instead of using normal brackets, it uses dollar brackets to embrace an expression. For example

```
...
      a = ${+ 1 1}
      b = ${<-> Phillip Ph F}
...
```

will result assign `"2"` to the variable `a` and `"Fillip"` to variable `b`. Dollar bracket expressions can, of course be nested and they allow conditional assignments. However, iteration, or even recursion is purposely not implemented. This is, in order to avoid possible unwanted infinite iterations/recursions caused by the writer of the configuration file[6]. An overview over all operators is provided in table 1. Please, note that you do not have to use any of those to write GetPot configuration files. Simply start using them when you need them.

11.1 String Operations

Table **??** lists the dollar bracket expressions that allow string operations: A dollar bracket expression

[6]The idea behind is that the responsibility for the functioning of an application shall lie on the programmer. He has to make sure that the program, either produces error/warning messages or functions properly. An infinite recursion in the configuration file could not be caught be the application. An undocumented malfunctioning however is not acceptable, since the user has no means to adapt his inputs. The author is well aware that there are major software companies that do not share this philosophy.

String operations	
`${string}`	variable replacement
`${:string}`	pure string (no parsing inside)
`${& string1 string2 string3 ...}`	concatenation
`${<-> string original replacement}`	string replacement
Arithmetic operations	
`${+ arg1 arg2 arg3 ...}`	plus
`${* arg1 arg2 arg3 ...}`	multiplication
`${- arg1 arg2 arg3 ...}`	subtraction
`${/ arg1 arg2 arg3 ...}`	division
`${^ arg1 arg2 arg3 ...}`	power
Comparisons	
`${== arg0 arg1 arg2 ...}`	equal
`${> arg0 arg1 arg2 ...}`	greater
`${< arg0 arg1 arg2 ...}`	less
`${>= arg0 arg1 arg2 ...}`	greater or equal
`${<= arg0 arg1 arg2 ...}`	less or equal
Conditions	
`${?  arg0 arg1 arg2}`	if-then
`${??  arg0 arg1 arg2 ...}`	choice
Vector/String subscriptions	
`${@:  string index0}`	specific letter in string
`${@:  string index0 index1}`	substring in string
`${@ variable index0}`	specific element in vector variable
`${@ variable index0 index1}`	sub-vector in vector variable
Macros	
`${!  string}`	macro expansion

Table 1: Total set of dollar bracket operators.

Table 2: String operations.

`${string}`	variable replacement
`${:string}`	pure string (no parsing inside)
`${& string1 string2 string3 ...}`	concatenation
`${<-> string original replacement}`	string replacement

that only contains a name is treated as variable expansion.

```
...
   name = GetPot
   [${name}] # meaning: [GetPot]
...
```

will set a section label `"GetPot"`.

```
...
   [Mechanical-Engineering]
      boss       = Dr.\ Frieda\ LaBlonde
      members    = 24
      professors = 5
   []
   x = Mechanical-Engineering

   info = '${${x}/boss}: ${${x}/professors}/${${x}/members}'
...
```

will assign the string `"Dr. Frieda La Blonde: 5/24"` to the variable `info`. Together with sections, the dollar bracket expressions allow an elegant way to define dictionaries, such as:

```
...
   my-car = Citroen-2CV
   [Nicknames]
        BMW          = Beamer
        Mercedez     = Grandpa\'s\ Slide
        Volkswagen   = Beetle
        Citroen-2CV  = Deuche
   []
   my-car = ${Nicknames/${my-car}}
...
```

uses the section `"Nicknames"` as dictionary. At the end the variable `my-car` will contain the name `"Deuche"` instead of `"Citroen-2CV"`.

Some users might miss the ability to have dollar bracket expressions or white spaces inside their strings, therefore a feature is provided that enables to specify pure strings. Anything in between a `${: ...}` environment is left as is without any modification. In the following example

```
...
   info = ${:even expressions like ${my-car} are left as they are}
...
```

whitespaces and brackets are left as they are. This feature is essential when defining macros **??**.

Strings can be concatenated by the `${& }`-operator as in the following example

```
info  = ${& simple concatination without whitespaces results in a mess}
```

As a result of this statement, `info` would contain the string

```
"simpleconcatinationwithoutwhitespacesresultsinamess".
```

in other words: if you want to form a sentence consisting of words separated by whitespaces, you should either use backslashed whitespaces, quotes or *pure strings*. As any other normal dollar bracket

expression it can contain dollar bracket expressions as part of its arguments such as in the following example:

```
name = FriedaBoelkenwater
network = neurology.west-wing.gov
contact-info = ${& ${:Email:} ${name} @ ${network}}
```

where the `contact-info` would be

```
"Email: FriedaBoelkenwater@neurology.west-wing.gov"
```

Another important string operation are replacements using the `${<->}`-operator such as in

```
OBJECTS = main.o tires.o suspension.o drive-train.o cvi.o steering-system.o
PROGRAM_FILES = ${<-> ${OBJECTS} .o .cpp}
```

where the extensions '.o' are replaced by extension '.cpp' to get the filenames of the source code.

11.2 Arithmetic Operations

Table **??** lists the dollar bracket expressions that allow arithmetic operations. The summation operator `${+ }` simply adds up all arguments and returns a string containing the total sum. Similarly, the multiplication operator multiplies all arguments as returns the total product.

The subtraction operator subtracts all arguments after the first argument from the first argument, i.e.

```
x = ${- 100 50 4 2}
y = ${/ 12 2 3}
```

assigns the value `45` to the variable `x`. Similarly, the division operator divides the first argument by all following arguments. Variable `y` therefore carries the value `2` after parsing.

Table 3: Arithmetic operations.

`${+ arg1 arg2 arg3 ...}`	plus
`${* arg1 arg2 arg3 ...}`	multiplication
`${- arg1 arg2 arg3 ...}`	subtraction
`${/ arg1 arg2 arg3 ...}`	division
`${^ arg1 arg2 arg3 ...}`	power

11.3 Comparison Operators

Table **??** lists the dollar bracket expressions that allow comparisons. These operators will return the number of the first argument for which the operator is true with respect to argument `arg0`. If none matches '0' is returned. Example:

```
country-id = ${== ${code} .de .fr .at .it .ch .cz}
```

will fill the variable country-id with `3` in case that `code` is `".at"`.

Table 4: Comparisons.

`${== arg0 arg1 arg2 ...}`	equal
`${> arg0 arg1 arg2 ...}`	greater
`${< arg0 arg1 arg2 ...}`	less
`${>= arg0 arg1 arg2 ...}`	greater or equal
`${<= arg0 arg1 arg2 ...}`	less or equal

11.4 Conditional Expansion

The operator in table 5 allow conditional expansion. The `${?  }`-operator constitutes a if-then statement. If `arg0`, usually the result of a comparison operation, is equal to 1 than `arg1` is returned - otherwise `arg2`. The `${??  }`-operator allows to choose from a list of arguments. If `arg0` is '1' than `arg1` is returned, if it is '2' than `arg2` is returned, etc.

Table 5: Conditional expansion.

`${?  arg0 arg1 arg2}`	if-then
`${??  arg0 arg1 arg2 ...}`	choice

11.5 Vector and String Subscription

The operator in table 6 allow vector and string subscriptions. A `${@:  }`-operator performs string subscription, a `${@ }`-operator performs vector subscription. If only one index is specified, than only one specific element is returned. If two indices are given, the substring/sub-vector from the first to the last index is returned. If the second index is '-1' it is considered to be equal to the size of the vector/string.

Table 6: Vector and string subscription.

`${@:  string index0}`	specific letter in string
`${@:  string index0 index1}`	substring in string
`${@ variable index0}`	specific element in vector variable
`${@ variable index0 index1}`	sub-vector in vector variable

11.6 Macro Calls

Any string stored in a variable can serve as a macro, so there is no need to have a macro-definition operator. The pure strings using the `${:  }`-operator come handy, though, since they allow to define dollar bracket expressions that are not directly parsed. Strings are parsed with the `${!  }`-operator. Defining

```
x2  = ${: ${* ${x} ${x}}}
x4  = ${: ${* ${!x2} ${!x2}}}
x6  = ${: ${* ${!x4} ${!x2}}}
sin = ${: ${* ${x}
```

```
                ${+ 1
                    ${/ ${!x2} -6}
                    ${/ ${!x4} 120}
                    ${/ ${!x6} -5040}
                }
            }
        }
```

makes it possible to compute sinuses inside a GetPot file, for example

```
x = 0.212
info = ${!sin}
```

will assign something that is close to the sinus of 0.212 to the variable *info*. Keep in mind, that through the huge amount of float-string conversions, back and forth, a lot of precision is lost. Don't consider this as a disadvantage ! As said before, configuration files are not there to write programs. A configuration file language, therefore, has to be a bit cumbersome, in order to prevent mayhem.