# User's Guide

# for the

# GNU GPERF Utility

Douglas C. Schmidt

last updated 14 April 1998

for version 2.7

# GNU GENERAL PUBLIC LICENSE

Version 1, February 1989



## Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.
5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is

copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

### NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## END OF TERMS AND CONDITIONS

## Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 1, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program 'Gnomovision' (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That's all there is to it!

# Contributors to GNU gperf Utility

- The GNU gperf perfect hash function generator utility was originally written in GNU C++ by Douglas C. Schmidt. It is now also available in a highly-portable "old-style" C version. The general idea for the perfect hash function generator was inspired by Keith Bostic's algorithm written in C, and distributed to net.sources around 1984. The current program is a heavily modified, enhanced, and extended implementation of Keith's basic idea, created at the University of California, Irvine. Bugs, patches, and suggestions should be reported to <bug-gnu-utils@gnu.org> and <schmidt@ics.uci.edu>.
- Special thanks is extended to Michael Tiemann and Doug Lea, for providing a useful compiler, and for giving me a forum to exhibit my creation.

  In addition, Adam de Boor and Nels Olson provided many tips and insights that greatly helped improve the quality and functionality of gperf.

# 1 Introduction

`gperf` is a perfect hash function generator written in C++. It transforms an $n$ element user-specified keyword set $W$ into a perfect hash function $F$. $F$ uniquely maps keywords in $W$ onto the range $0..k$, where $k >= n$. If $k = n$ then $F$ is a *minimal* perfect hash function. `gperf` generates a $0..k$ element static lookup table and a pair of C functions. These functions determine whether a given character string $s$ occurs in $W$, using at most one probe into the lookup table.

`gperf` currently generates the reserved keyword recognizer for lexical analyzers in several production and research compilers and language processing tools, including GNU C, GNU C++, GNU Pascal, GNU Modula 3, and GNU indent. Complete C++ source code for `gperf` is available via anonymous ftp from `ics.uci.edu` and `ftp.santafe.edu`. `gperf` was also distributed along with the GNU libg++ library for several years. A highly portable, functionally equivalent K&R C version of `gperf` is archived in comp.sources.unix, volume 20. Finally, a paper describing `gperf`'s design and implementation in greater detail is available in the Second USENIX C++ Conference proceedings.

# 2 Static search structures and GNU `gperf`

A *static search structure* is an Abstract Data Type with certain fundamental operations, e.g., *initialize*, *insert*, and *retrieve*. Conceptually, all insertions occur before any retrievals. In practice, `gperf` generates a `static` array containing search set keywords and any associated attributes specified by the user. Thus, there is essentially no execution-time cost for the insertions. It is a useful data structure for representing *static search sets*. Static search sets occur frequently in software system applications. Typical static search sets include compiler reserved words, assembler instruction opcodes, and built-in shell interpreter commands. Search set members, called *keywords*, are inserted into the structure only once, usually during program initialization, and are not generally modified at run-time.

Numerous static search structure implementations exist, e.g., arrays, linked lists, binary search trees, digital search tries, and hash tables. Different approaches offer trade-offs between space utilization and search time efficiency. For example, an $n$ element sorted array is space efficient, though the average-case time complexity for retrieval operations using binary search is proportional to $\log n$. Conversely, hash table implementations often locate a table entry in constant time, but typically impose additional memory overhead and exhibit poor worst case performance.

*Minimal perfect hash functions* provide an optimal solution for a particular class of static search sets. A minimal perfect hash function is defined by two properties:

- It allows keyword recognition in a static search set using at most *one* probe into the hash table. This represents the "perfect" property.
- The actual memory allocated to store the keywords is precisely large enough for the keyword set, and *no larger*. This is the "minimal" property.

For most applications it is far easier to generate *perfect* hash functions than *minimal perfect* hash functions. Moreover, non-minimal perfect hash functions frequently execute faster than minimal ones in practice. This phenomena occurs since searching a sparse keyword table increases the probability of locating a "null" entry, thereby reducing string comparisons. `gperf`'s default behavior generates *near-minimal* perfect hash functions for keyword sets. However, `gperf` provides many options that permit user control over the degree of minimality and perfection.

Static search sets often exhibit relative stability over time. For example, Ada's 63 reserved words have remained constant for nearly a decade. It is therefore frequently worthwhile to expend concerted effort building an optimal search structure *once*, if it subsequently receives heavy use multiple times. `gperf` removes the drudgery associated with constructing time- and space-efficient search structures by hand. It has proven a useful and practical tool for serious programming projects. Output from `gperf` is currently used in several production and research compilers, including GNU C, GNU C++, GNU Pascal, and GNU Modula 3. The latter two compilers are not yet part of the official GNU distribution. Each compiler utilizes `gperf` to automatically generate static search structures that efficiently identify their respective reserved keywords.

# 3 High-Level Description of GNU gperf

The perfect hash function generator gperf reads a set of "keywords" from a *keyfile* (or from the standard input by default). It attempts to derive a perfect hashing function that recognizes a member of the *static keyword set* with at most a single probe into the lookup table. If gperf succeeds in generating such a function it produces a pair of C source code routines that perform hashing and table lookup recognition. All generated C code is directed to the standard output. Command-line options described below allow you to modify the input and output format to gperf.

By default, gperf attempts to produce time-efficient code, with less emphasis on efficient space utilization. However, several options exist that permit trading-off execution time for storage space and vice versa. In particular, expanding the generated table size produces a sparse search structure, generally yielding faster searches. Conversely, you can direct gperf to utilize a C switch statement scheme that minimizes data space storage size. Furthermore, using a C switch may actually speed up the keyword retrieval time somewhat. Actual results depend on your C compiler, of course.

In general, gperf assigns values to the characters it is using for hashing until some set of values gives each keyword a unique value. A helpful heuristic is that the larger the hash value range, the easier it is for gperf to find and generate a perfect hash function. Experimentation is the key to getting the most from gperf.

## 3.1 Input Format to gperf

You can control the input keyfile format by varying certain command-line arguments, in particular the '-t' option. The input's appearance is similar to GNU utilities flex and bison (or UNIX utilities lex and yacc). Here's an outline of the general format:

```
declarations
%%
keywords
%%
functions
```

*Unlike* flex or bison, all sections of gperf's input are optional. The following sections describe the input format for each section.

### 3.1.1 struct Declarations and C Code Inclusion

The keyword input file optionally contains a section for including arbitrary C declarations and definitions, as well as provisions for providing a user-supplied `struct`. If the '`-t`' option *is* enabled, you *must* provide a C `struct` as the last component in the declaration section from the keyfile file. The first field in this struct must be a `char *` identifier called '`name`', although it is possible to modify this field's name with the '`-K`' option described below.

Here is simple example, using months of the year and their attributes as input:

```
struct months { char *name; int number; int days; int leap_days; };
%%
january,   1, 31, 31
february,  2, 28, 29
march,     3, 31, 31
april,     4, 30, 30
may,       5, 31, 31
june,      6, 30, 30
july,      7, 31, 31
august,    8, 31, 31
september, 9, 30, 30
october,  10, 31, 31
november, 11, 30, 30
december, 12, 31, 31
```

Separating the `struct` declaration from the list of key words and other fields are a pair of consecutive percent signs, `%%`, appearing left justified in the first column, as in the UNIX utility `lex`.

Using a syntax similar to GNU utilities `flex` and `bison`, it is possible to directly include C source text and comments verbatim into the generated output file. This is accomplished by enclosing the region inside left-justified surrounding `%{`, `%}` pairs. Here is an input fragment based on the previous example that illustrates this feature:

```
%{
#include <assert.h>
/* This section of code is inserted directly into the output. */
int return_month_days (struct months *months, int is_leap_year);
%}
struct months { char *name; int number; int days; int leap_days; };
%%
january,   1, 31, 31
february,  2, 28, 29
march,     3, 31, 31
...
```

It is possible to omit the declaration section entirely. In this case the keyfile begins directly with the first keyword line, e.g.:

```
january,   1, 31, 31
february,  2, 28, 29
march,     3, 31, 31
april,     4, 30, 30
...
```

### 3.1.2 Format for Keyword Entries

The second keyfile format section contains lines of keywords and any associated attributes you might supply. A line beginning with '#' in the first column is considered a comment. Everything following the '#' is ignored, up to and including the following newline.

The first field of each non-comment line is always the key itself. It should be given as a simple name, i.e., without surrounding string quotation marks, and be left-justified flush against the first column. In this context, a "field" is considered to extend up to, but not include, the first blank, comma, or newline. Here is a simple example taken from a partial list of C reserved words:

```
# These are a few C reserved words, see the c.gperf file
# for a complete list of ANSI C reserved words.
unsigned
sizeof
switch
signed
if
default
for
while
return
```

Note that unlike flex or bison the first %% marker may be elided if the declaration section is empty.

Additional fields may optionally follow the leading keyword. Fields should be separated by commas, and terminate at the end of line. What these fields mean is entirely up to you; they are used to initialize the elements of the user-defined struct provided by you in the declaration section. If the '-t' option is *not* enabled these fields are simply ignored. All previous examples except the last one contain keyword attributes.

### 3.1.3 Including Additional C Functions

The optional third section also corresponds closely with conventions found in flex and bison. All text in this section, starting at the final %% and extending to the end of the input file, is included verbatim into the generated output file. Naturally, it is your responsibility to ensure that the code contained in this section is valid C.

## 3.2 Output Format for Generated C Code with gperf

Several options control how the generated C code appears on the standard output. Two C function are generated. They are called hash and in_word_set, although you may modify the name for in_word_set with a command-line option. Both functions require two arguments, a string, char * *str*, and a length parameter, int *len*. Their default function prototypes are as follows:

```
static int hash (char *str, int len);
int in_word_set (char *str, int len);
```

By default, the generated hash function returns an integer value created by adding *len* to several user-specified *str* key positions indexed into an *associated values* table stored in a local static array. The associated values table is constructed internally by gperf and later output as a static local C array called *hash_table*; its meaning and properties are described below. See Chapter 7 [Implementation], page 29. The relevant key positions are specified via the '-k' option when running gperf, as detailed in the *Options* section below. See Chapter 4 [Options], page 19.

Two options, '-g' (assume you are compiling with GNU C and its inline feature) and '-a' (assume ANSI C-style function prototypes), alter the content of both the generated hash and in_ word_set routines. However, function in_word_set may be modified more extensively, in response to your option settings. The options that affect the in_word_set structure are:

'-t' Make use of the user-defined struct.

'-S *total switch statements*'
Generate 1 or more C switch statement rather than use a large, (and potentially sparse) static array. Although the exact time and space savings of this approach vary according to your C compiler's degree of optimization, this method often results in smaller and faster code.

If the '-t' and '-S' options are omitted, the default action is to generate a char * array containing the keys, together with additional null strings used for padding the array. By experimenting with the various input and output options, and timing the resulting C code, you can determine the best option choices for different keyword set characteristics.

# 4 Options to the gperf Utility

There are *many* options to `gperf`. They were added to make the program more convenient for use with real applications. "On-line" help is readily available via the '`-h`' option. Here is the complete list of options.

## 4.1 Options that affect Interpretation of the Input File

'`-e` *keyword delimiter list*'
: Allows the user to provide a string containing delimiters used to separate keywords from their attributes. The default is ",\n". This option is essential if you want to use keywords that have embedded commas or newlines. One useful trick is to use -e'TAB', where TAB is the literal tab character.

'`-t`'
: Allows you to include a `struct` type declaration for generated code. Any text before a pair of consecutive %% is consider part of the type declaration. Key words and additional fields may follow this, one group of fields per line. A set of examples for generating perfect hash tables and functions for Ada, C, and G++, Pascal, and Modula 2 and 3 reserved words are distributed with this release.

## 4.2 Options to specify the Language for the Output Code

'`-L` *generated language name*'
: Instructs `gperf` to generate code in the language specified by the option's argument. Languages handled are currently:

  '`KR-C`'
  : Old-style K&R C. This language is understood by old-style C compilers and ANSI C compilers, but ANSI C compilers may flag warnings (or even errors) because of lacking '`const`'.

  '`C`'
  : Common C. This language is understood by ANSI C compilers, and also by old-style C compilers, provided that you `#define const` to empty for compilers which don't know about this keyword.

  '`ANSI-C`'
  : ANSI C. This language is understood by ANSI C compilers and C++ compilers.

  '`C++`'
  : C++. This language is understood by C++ compilers.

The default is C.

'-a' This option is supported for compatibility with previous releases of gperf. It does not do anything.

'-g' This option is supported for compatibility with previous releases of gperf. It does not do anything.

## 4.3 Options for fine tuning Details in the Output Code

'-K *key name*'
: This option is only useful when option '-t' has been given. By default, the program assumes the structure component identifier for the keyword is 'name'. This option allows an arbitrary choice of identifier for this component, although it still must occur as the first field in your supplied struct.

'-H *hash function name*'
: Allows you to specify the name for the generated hash function. Default name is 'hash'. This option permits the use of two hash tables in the same file.

'-N *lookup function name*'
: Allows you to specify the name for the generated lookup function. Default name is 'in_word_set'. This option permits completely automatic generation of perfect hash functions, especially when multiple generated hash functions are used in the same application.

'-Z *class name*'
: This option is only useful when option '-L C++' has been given. It allows you to specify the name of generated C++ class. Default name is Perfect_Hash.

'-7' This option specifies that all strings that will be passed as arguments to the generated hash function and the generated lookup function will solely consist of 7-bit ASCII characters (characters in the range 0..127). (Note that the ANSI C functions isalnum and isgraph do *not* guarantee that a character is in this range. Only an explicit test like 'c >= 'A' && c <= 'Z'' guarantees this.) This was the default in earlier versions of gperf; now the default is to assume 8-bit characters.

'-c' Generates C code that uses the strncmp function to perform string comparisons. The default action is to use strcmp.

'-C' Makes the contents of all generated lookup tables constant, i.e., "readonly". Many compilers can generate more efficient code for this by putting the tables in readonly memory.

'`-E`' Define constant values using an enum local to the lookup function rather than with #defines. This also means that different lookup functions can reside in the same file. Thanks to James Clark `<jjc@ai.mit.edu>`.

'`-I`' Include the necessary system include file, `<string.h>`, at the beginning of the code. By default, this is not done; the user must include this header file himself to allow compilation of the code.

'`-G`' Generate the static table of keywords as a static global variable, rather than hiding it inside of the lookup function (which is the default behavior).

'`-W` *hash table array name*'
: Allows you to specify the name for the generated array containing the hash table. Default name is '`wordlist`'. This option permits the use of two hash tables in the same file, even when the option '`-G`' is given.

'`-S` *total switch statements*'
: Causes the generated C code to use a `switch` statement scheme, rather than an array lookup table. This can lead to a reduction in both time and space requirements for some keyfiles. The argument to this option determines how many `switch` statements are generated. A value of 1 generates 1 `switch` containing all the elements, a value of 2 generates 2 tables with 1/2 the elements in each `switch`, etc. This is useful since many C compilers cannot correctly generate code for large `switch` statements. This option was inspired in part by Keith Bostic's original C program.

'`-T`' Prevents the transfer of the type declaration to the output file. Use this option if the type is already defined elsewhere.

'`-p`' This option is supported for compatibility with previous releases of `gperf`. It does not do anything.

## 4.4 Options for changing the Algorithms employed by `gperf`

'`-k` *keys*' Allows selection of the character key positions used in the keywords' hash function. The allowable choices range between 1-126, inclusive. The positions are separated by commas, e.g., '`-k 9,4,13,14`'; ranges may be used, e.g., '`-k 2-7`'; and positions may occur in any order. Furthermore, the meta-character '*' causes the generated hash function to consider **all** character positions in each key, whereas '$' instructs the hash function to use the "final character" of a key (this is the only way to use a character position greater than 126, incidentally).

For instance, the option '`-k 1,2,4,6-10,'$'`' generates a hash function that considers positions 1,2,4,6,7,8,9,10, plus the last character in each key (which may differ for each key, obviously). Keys with length less than the indicated key positions work properly, since selected key positions exceeding the key length are simply not referenced in the hash function.

'-l' Compare key lengths before trying a string comparison. This might cut down on the number of string comparisons made during the lookup, since keys with different lengths are never compared via `strcmp`. However, using '-l' might greatly increase the size of the generated C code if the lookup table range is large (which implies that the switch option '-S' is not enabled), since the length table contains as many elements as there are entries in the lookup table.

'-D' Handle keywords whose key position sets hash to duplicate values. Duplicate hash values occur for two reasons:

- Since `gperf` does not backtrack it is possible for it to process all your input keywords without finding a unique mapping for each word. However, frequently only a very small number of duplicates occur, and the majority of keys still require one probe into the table.
- Sometimes a set of keys may have the same names, but possess different attributes. With the -D option `gperf` treats all these keys as part of an equivalence class and generates a perfect hash function with multiple comparisons for duplicate keys. It is up to you to completely disambiguate the keywords by modifying the generated C code. However, `gperf` helps you out by organizing the output.

Option '-D' is extremely useful for certain large or highly redundant keyword sets, e.g., assembler instruction opcodes. Using this option usually means that the generated hash function is no longer perfect. On the other hand, it permits `gperf` to work on keyword sets that it otherwise could not handle.

'-f *iteration amount*'
Generate the perfect hash function "fast". This decreases `gperf`'s running time at the cost of minimizing generated table-size. The iteration amount represents the number of times to iterate when resolving a collision. '0' means iterate by the number of keywords. This option is probably most useful when used in conjunction with options '-D' and/or '-S' for *large* keyword sets.

'-i *initial value*'
Provides an initial *value* for the associate values array. Default is 0. Increasing the initial value helps inflate the final table size, possibly leading to more time efficient keyword lookups. Note that this option is not particularly useful when '-S' is used. Also, '-i' is overriden when the '-r' option is used.

'-j *jump value*'
: Affects the "jump value", i.e., how far to advance the associated character value upon collisions. *Jump value* is rounded up to an odd number, the default is 5. If the *jump value* is 0 `gperf` jumps by random amounts.

'-n'
: Instructs the generator not to include the length of a keyword when computing its hash value. This may save a few assembly instructions in the generated lookup table.

'-o'
: Reorders the keywords by sorting the keywords so that frequently occuring key position set components appear first. A second reordering pass follows so that keys with "already determined values" are placed towards the front of the keylist. This may decrease the time required to generate a perfect hash function for many keyword sets, and also produce more minimal perfect hash functions. The reason for this is that the reordering helps prune the search time by handling inevitable collisions early in the search process. On the other hand, if the number of keywords is *very* large using '-o' may *increase* `gperf`'s execution time, since collisions will begin earlier and continue throughout the remainder of keyword processing. See Cichelli's paper from the January 1980 Communications of the ACM for details.

'-r'
: Utilizes randomness to initialize the associated values table. This frequently generates solutions faster than using deterministic initialization (which starts all associated values at 0). Furthermore, using the randomization option generally increases the size of the table. If `gperf` has difficultly with a certain keyword set try using '-r' or '-D'.

'-s *size-multiple*'
: Affects the size of the generated hash table. The numeric argument for this option indicates "how many times larger or smaller" the maximum associated value range should be, in relationship to the number of keys. If the *size-multiple* is negative the maximum associated value is calculated by *dividing* it into the total number of keys. For example, a value of 3 means "allow the maximum associated value to be about 3 times larger than the number of input keys".

  Conversely, a value of -3 means "allow the maximum associated value to be about 3 times smaller than the number of input keys". Negative values are useful for limiting the overall size of the generated hash table, though this usually increases the number of duplicate hash values.

  If 'generate switch' option '-S' is *not* enabled, the maximum associated value influences the static array table size, and a larger table should decrease the time required for an unsuccessful search, at the expense of extra table space.

  The default value is 1, thus the default maximum associated value about the same size as the number of keys (for efficiency, the maximum associated value is always rounded up to a power of 2). The actual table size may vary somewhat, since this technique is essentially a heuristic. In particular, setting this value too high slows down `gperf`'s runtime, since it must search through a much larger range of values. Judicious use of the '`-f`' option helps alleviate this overhead, however.

## 4.5 Informative Output

'`-h`' Prints a short summary on the meaning of each program option. Aborts further program execution.

'`-v`' Prints out the current version number.

'`-d`' Enables the debugging option. This produces verbose diagnostics to "standard error" when `gperf` is executing. It is useful both for maintaining the program and for determining whether a given set of options is actually speeding up the search for a solution. Some useful information is dumped at the end of the program when the '`-d`' option is enabled.

# 5 Known Bugs and Limitations with `gperf`

The following are some limitations with the current release of `gperf`:

- The `gperf` utility is tuned to execute quickly, and works quickly for small to medium size data sets (around 1000 keywords). It is extremely useful for maintaining perfect hash functions for compiler keyword sets. Several recent enhancements now enable `gperf` to work efficiently on much larger keyword sets (over 15,000 keywords). When processing large keyword sets it helps greatly to have over 8 megs of RAM.

  However, since `gperf` does not backtrack no guaranteed solution occurs on every run. On the other hand, it is usually easy to obtain a solution by varying the option parameters. In particular, try the '`-r`' option, and also try changing the default arguments to the '`-s`' and '`-j`' options. To *guarantee* a solution, use the '`-D`' and '`-S`' options, although the final results are not likely to be a *perfect* hash function anymore! Finally, use the '`-f`' option if you want `gperf` to generate the perfect hash function *fast*, with less emphasis on making it minimal.
- The size of the generate static keyword array can get *extremely* large if the input keyword file is large or if the keywords are quite similar. This tends to slow down the compilation of the generated C code, and *greatly* inflates the object code size. If this situation occurs, consider using the '`-S`' option to reduce data size, potentially increasing keyword recognition time a negligible amount. Since many C compilers cannot correctly generated code for large switch statements it is important to qualify the *-S* option with an appropriate numerical argument that controls the number of switch statements generated.
- The maximum number of key positions selected for a given key has an arbitrary limit of 126. This restriction should be removed, and if anyone considers this a problem write me and let me know so I can remove the constraint.

# 6 Things Still Left to Do

It should be "relatively" easy to replace the current perfect hash function algorithm with a more exhaustive approach; the perfect hash module is essential independent from other program modules. Additional worthwhile improvements include:

- Make the algorithm more robust. At present, the program halts with an error diagnostic if it can't find a direct solution and the '`-D`' option is not enabled. A more comprehensive, albeit computationally expensive, approach would employ backtracking or enable alternative options and retry. It's not clear how helpful this would be, in general, since most search sets are rather small in practice.
- Another useful extension involves modifying the program to generate "minimal" perfect hash functions (under certain circumstances, the current version can be rather extravagant in the generated table size). Again, this is mostly of theoretical interest, since a sparse table often produces faster lookups, and use of the '`-S`' `switch` option can minimize the data size, at the expense of slightly longer lookups (note that the gcc compiler generally produces good code for `switch` statements, reducing the need for more complex schemes).
- In addition to improving the algorithm, it would also be useful to generate a C++ class or Ada package as the code output, in addition to the current C routines.

# 7 Implementation Details of GNU `gperf`

A paper describing the high-level description of the data structures and algorithms used to implement `gperf` will soon be available. This paper is useful not only from a maintenance and enhancement perspective, but also because they demonstrate several clever and useful programming techniques, e.g., 'Iteration Number' boolean arrays, double hashing, a "safe" and efficient method for reading arbitrarily long input from a file, and a provably optimal algorithm for simultaneously determining both the minimum and maximum elements in a list.

# 8 Bibliography

[1] Chang, C.C.: *A Scheme for Constructing Ordered Minimal Perfect Hashing Functions* Information Sciences 39(1986), 187-195.

[2] Cichelli, Richard J. *Author's Response to "On Cichelli's Minimal Perfect Hash Functions Method"* Communications of the ACM, 23, 12(December 1980), 729.

[3] Cichelli, Richard J. *Minimal Perfect Hash Functions Made Simple* Communications of the ACM, 23, 1(January 1980), 17-19.

[4] Cook, C. R. and Oldehoeft, R.R. *A Letter Oriented Minimal Perfect Hashing Function* SIGPLAN Notices, 17, 9(September 1982), 18-27.

[5] Cormack, G. V. and Horspool, R. N. S. and Kaiserwerth, M. *Practical Perfect Hashing* Computer Journal, 28, 1(January 1985), 54-58.

[6] Jaeschke, G. *Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions* Communications of the ACM, 24, 12(December 1981), 829-833.

[7] Jaeschke, G. and Osterburg, G. *On Cichelli's Minimal Perfect Hash Functions Method* Communications of the ACM, 23, 12(December 1980), 728-729.

[8] Sager, Thomas J. *A Polynomial Time Generator for Minimal Perfect Hash Functions* Communications of the ACM, 28, 5(December 1985), 523-532

[9] Schmidt, Douglas C. *GPERF: A Perfect Hash Function Generator* Second USENIX C++ Conference Proceedings, April 1990.

[10] Sebesta, R.W. and Taylor, M.A. *Minimal Perfect Hash Functions for Reserved Word Lists* SIGPLAN Notices, 20, 12(September 1985), 47-53.

[11] Sprugnoli, R. *Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets* Communications of the ACM, 20 11(November 1977), 841-850.

[12] Stallman, Richard M. *Using and Porting GNU CC* Free Software Foundation, 1988.

[13] Stroustrup, Bjarne *The C++ Programming Language.* Addison-Wesley, 1986.

[14] Tiemann, Michael D. *User's Guide to GNU C++* Free Software Foundation, 1989.

# Table of Contents