Genparse

Command Line Parser Generator
Edition 0.9.3, for Genparse version 0.9.3
13 November 2016

Mike Borella, Michael Geng

Table of Contents

1 Introduction

The Genparse package allows the automated creation of command line parsing routines, based on a command line option description file. Given a target language, Genparse outputs a suitable parser. For example, in C, a function is created that reads the command line parameters and stores their values in a data structure. In C++ and Java, a command line parsing class is created, with member functions that return parameter values.

The goal of Genparse is to eliminate the effort required to write a command line parser for every newly-developed program. Typically, these parsers all follow a very similar format, with a series of calls to the `getopt ()` or `getopt_long ()` library functions. Genparse allows users to specify their program's command line options in a concise text file (hereafter, a "Genparse File"). Genparse reads this file, and produces the appropriate parsing routines in the user's language of choice. Users may call the parser's functions from their main code in order to obtain the values of the parameters.

In addition to providing a simple interface to the parameters, Genparse also has the following features:

- A default value may be provided for each parameter.
- Both short (`-o`) and long (`--option`) parameters are supported.
- For parameters that take numerical values, Genparse can make sure that the input values fall within a given range.
- A `usage ()` function is automatically created, which can be used to describe a program's command line parameters. Genparse also allows a description string to be associated with each parameter. These strings will be displayed in the `usage ()` function.
- For C and C++ extra include files may be added to any command line parser.
- Each parameter can have a callback function associated with it. Genparse will automatically create a skeleton for callback functions, which the user can fill in. Also, global callback functions can be specified.
- The generated parser is 100% GNU compatible because the parsing itself is done by the GNU getopt_long function.
- The generated code is internationalizable (See [internationalize], page 29.).

Currently, we do not believe that Genparse has any significant limitations. It cannot handle dependencies between different options, but the user can write callback function to accomplish this task. A virtually unlimited number of long command line options are available, but only 52 single-character options can be used by a given program.

Mail suggestions and bug reports for Genparse to `mike@borella.net`. The latest version of Genparse can always be found at `http://genparse.sourceforge.net`.

2 Making Genparse Files

In this section we discuss how to write Genparse files and how to invoke Genparse on these files. We take a pedagogical approach, walking through a very simple example program and showing how Genparse can simplify its development.

2.1 A Simple Application

Suppose that we want to write a C program that outputs a given text file some number of times. This is a simple, and perhaps not terribly useful program, but its simplicity will help illustrate the utility of Genparse.

Our program, `mycopy1`, might look like this:

```
/* mycopy1.c */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int c, i, n;
  FILE *fp;

  n = atoi (argv[1]);
  fp = fopen (argv[2],"r");

  for (i = 0; i < n; i++)
    {
      while ((c = fgetc (fp)) != EOF)
fputc (c, stdout);
      rewind (fp);
    }

  fclose (fp);
  return 0;
}
```

The user is expected to invoke `mycopy1` with two items on the command line: an integer followed by a filename. The former is the number of times that the latter should be displayed. While this program accomplishes what we set out to do, it is not very robust nor user friendly. For example, if the user specifies a negative integer, the program does not display a warning or error message (in fact, it displays nothing). If the user does not know what is expected on the command line, how will he or she find this information out (assuming that nice documentation, such as what you are now reading, does not exist for `mycopy1`). Furthermore, wouldn't this program be more flexible if there were a default number of iterations, the user could specify the command line parameters in any order or omit some altogether?

All of these issues, and perhaps others, can be addressed in a number of ways. Traditionally, the author of `mycopy1` would publish the command line format, typically in a man page, and write a routine to pull the command line parameters out of the argv array, assuming that the format was followed (not unlike what we've done for mycopy1). However, as the number of command line parameters increases, this task becomes much more difficult and cumbersome.

With the introduction of the getopt () and getopt_long () functions, now part of the GNU C Library, a great deal of the command line parsing burden was lifted from programmers. The getopt () function takes in an argv-style command line and assumes that it contains a series of command line options. An option is indicated with a single character preceded by a dash - for example, '-o' or '-Z'. These options may be followed by a command line parameter - for example, '-o' or '-Z 3'. Using getopt we could add an '-i' to allow the number of iterations to be specified. The advantage to doing so is that it would no longer matter where on the command line '-i' appears, and if '-i' does not appear at all, we can assign a default number of iterations.

The getopt_long () function extends getopt (), by allowing long options to coexist with single character options. Long options are preceded by two dashes and may be more than one character long - for example '--iterations'. Long options may also take parameters, in the form '--option param' or '--option=param'.

2.2 Adding Command Line Options

From the previous section's mycopy1, we will now add command line option parsing with getopt_long () to create mycopy2. Along the way we'll add a small number of useful features.

```
/* mycopy2.c */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main (int argc, char *argv[])
{
  int c, i;
  FILE *fp;
  extern char *optarg;
  extern int optind;
  int option_index = 0;
  char ch;
  int help, iterations;
  int errflg = 0;

  static struct option long_options[] =
  {
    {"help", no_argument, NULL, 'h'},
    {"iterations", required_argument, NULL, 'i'},
    {NULL, 0, NULL, 0}
  };
  help = 0;
  iterations = 1;

  while ((ch = getopt_long (argc, argv, "hi:", long_options,
    &option_index)) != EOF)
    {
      switch (ch)
        {
case 'h':
          help = 1;
          break;

        case 'i':
          iterations = atoi (optarg);
          if (iterations < 0)
            {
              fprintf (stderr, "error: iterations must be >= 0\n");
              errflg ++;
            }
          break;

        default:
          errflg++;

}
    } /* while */

  if (errflg || help)
    {
      printf ("usage: %s [ -i ] <file>\n", argv[0]);
      printf ("  [ -h ] [ --help ] Print help screen \n");
      printf ("  [ -i ] [ --iterations ] Number of times to \
output <file>\n");
      exit (1);
    }

  if (optind >= argc)
    fp = stdin;
  else
    fp = fopen (argv[optind],"r");

  for (i = 0; i < iterations; i++)
    {
      while ((c = fgetc (fp)) != EOF)
fputc (c, stdout);
      rewind (fp);
    }
  fclose (fp);
  return 0;
}
```

This program performs the same function as `mycopy1` but does so in a more flexible and reliable fashion. Two command line options are supported, '`-h`' and '`-i`'. When '`-h`', or its long form, '`--help`', appears on the command line, all other options are ignored and a usage message is displayed. The '`-i`' option allows the user to specify the number of iterations, as discussed above. It also has a long form, '`--iterations`'. The number of iterations defaults to 1 if '`-i`' does not appear on the command line, and the program prevents negative values from being specified with '`-i`'.

The usage message is a useful way of summarizing a program's options without needed a `man` or `info` page. There are three ways that the usage message for `mycopy2` will be displayed:

- If the '`-h`' or '`--help`' options appear on the command line.
- If an unknown option appears on the command line.
- If the '`-i`' option appears with a negative value.

A nice feature of `getopt ()` and `getopt_long ()` is that they will rearrange the command line within `argv` so that all non-options follow all of the options[1]. The external variable `optind` is set to point to the first non-option in the re-arranged `argv`. Thus, by comparing `optind` to `argc`, we can determine whether or not an input file has been specified. If there is no input file on the command line, we can redirect `mycopy2` to use *stdin*.

While `mycopy2` is an improvement over `mycopy1`, there are a number of drawbacks to using `getopt ()` in all of your programs. The most obvious is time. In `mycopy2`, two-thirds (about 50 lines) of the code does the command line parsing, and only two options are supported. If there we're 10 options, the command line parsing code could easily reach 200 lines or more. Additionally, since each option's parameter may need to be checked for validity and assigned to a variable, this becomes a tedious process that can be error prone.

Observing the source code for `mycopy2`, it becomes clear that, like any repetitive task, the command line parsing code follows a number of simple patterns. If these patterns can be abstracted and generalized so that the user can indicate option use in a concise format, most, if not all, of the parsing code can be automatically generated. With this thought in mind, we turn our attention to Genparse.

2.3 Simplifying Command Line Parsing with Genparse

Genparse automatically creates command line parsing code, not unlike the code in `mycopy2.c`. It creates two or three files: a header file, a parsing file, and an optional callback file. In this section, we'll write a Genparse specification for our file copying program and examine the parser that it creates.

Genparse runs on a simple input file, which we'll call a Genparse file. In a Genparse file, each command line option is specified on one or more lines. The following code is a Genparse file for `mycopy3`.

[1] `argv[0]` is not rearranged - it remains in its place.

```
/* mycopy3.gp */
i / iterations  int 1 [0...]  "Number of times to output <file>."
"File should be text format!"
o / outfile string {""} "Output file name."

#usage_begin
usage: __PROGRAM_NAME__ __OPTIONS_SHORT__ file
Print a file for a number of times to stdout.
__GLOSSARY__
#usage_end
```

The naming of this file follows the convention of all Genparse files ending in the extension `.gp`.

Let's walk through `mycopy3.gp`. The first line is a comment. It is ignored by Genparse.

The second line is the first option specification. This is the '`-i`' option, which, as before, may be specified in long form as '`--iterations`'. Our `mycopy3.gp` file indicates that '`-i`' must take an integer parameter, the default value of which is 1. The allowed range is non-negative. The final part of the third line is a description of the option's usage (to appear in the `usage ()` function).

The third line introduces a new option, that was not in `mycopy2`. The '`-o`' option takes a string parameter (where a "string" is any series of characters) with a default value of empty. As indicated by the description, this option is used to specify an output file to which `mycopy3`'s output will be directed.

Default values for strings must be specified within braces and quotes like `{"`This is a stupid comment`"}`, for chars it must be enclosed in single quotes, e.g. 'a' or '\0x13'. For other integers use the plain default value.

Starting in the third line the help screen is defined which will be printed if '`-h`' or '`--help`' is set or if an invalid command line is given. `__PROGRAM_NAME__` will be replaced with the name of the executable (probably mycopy3), `__OPTIONS_SHORT__` will be replaced with a list of allowed short options ("[-iohv]" in this example). "Print a file for a number of times to stdout." will be printed verbatim. `__GLOSSARY__` will be replaced with a list explanations for each of the command line parameters. For more explanations on the `#usage` section See Section 4.3 [Usage Function], page 33. `mycopy3 --help` will print the following help screen:

```
usage: mycopy3 [ -iohv ] file
Print file for a number of times to stdout.
   [ -i ] [ --iterations ] (type=INTEGER, range=0..., default=1)
          Number of times to output <file>.
          File should be text format!
   [ -o ] [ --outfile ] (type=STRING)
          Output file name.
   [ -h ] [ --help ] (type=FLAG)
          Display this help and exit.
   [ -v ] [ --version ] (type=FLAG)
          Output version information and exit.
```

Genparse can be invoked on mycopy3.gp in a number of ways (Yes, Genparse has its own command line options! See Chapter 3 [Genparse Options], page 29.), but we'll invoke it as follows.

```
genparse -o mycopy3_clp mycopy3.gp
```

This command tells Genparse to run on mycopy3.gp and to output program files named with mycopy3_clp. This particular Genparse file creates only a header file and a parser file since no callbacks are specified. Let's first take a look at the header file, mycopy3_clp.h.

2.3.1 Header Files

Below, we walk through mycopy3_clp.h, an example header file created by Genparse. Header files, such as this one, *must* be included in all linked code that needs to access the command line parameter values.

```
/* mycopy3_clp.h */

#include <stdio.h>

#ifndef bool
typedef enum bool_t
{
  false = 0, true
} bool;
#endif

/* customized structure for command line parameters */
struct arg_t
{
  int i;
  char * o;
  bool h;
  bool v;
  int optind;
};

/* function prototypes */
void Cmdline (struct arg_t *my_args, int argc, char *argv[]);
void usage (int status, char *program_name);
```

Although "real" Genparse output files begin with a section of comments, for purposes of saving space, we'll replace all of those with a short comment containing only the file's name.

A Genparse-created header file contains four major sections: (1) includes and type definitions, (2) the definition of struct arg_t, (3) parsing function prototypes, and (4) callback function prototypes. Since we are not using callbacks in this example, only the first three sections appear in this header file.

The file begins with a list of header files to include. Including stdio.h is the default, and other includes may be specified in the Genparse file. Then the bool type is conditionally defined. While `bool` is typically predefined in C++, it is not in C. It comes in handy as a type for all flag options, which can only be on or off (true or false).

The `struct arg_t` structure contains a variable for each of the options defined in `mycopy3.gp`. This include `i`, an integer, and `o`, a character pointer. For C output, all variables defined to be strings in Genparse files are declared as character pointers. For C++ output, the C++ string type from the standard C++ library is used.

In addition to the user-defined options, Genparse adds two extra flag options, '`-h`' and '`-v`'. The '`-h`' option (long form of '`--help`') will cause the `usage ()` function to be called, and the program to terminate. The '`-v`' option (long form of '`--version`') will be passed back for the calling function to process. It is intended that the caller will display the program's version number if this option is set. Note that if the calling program does not process the '`-v`' flag, its behavior will not be affected by this flag.

The `optind` variable records the value of the `optind` static variable that is used by `getopt ()` and `getopt_long ()`. However, Genparse has changed the behavior of this variable slightly. (See Section 2.3.2 [Parser Files], page 8.)

The final section of `mycopy3_clp.h` consists of function prototypes for the command line parser `Cmdline ()`[2], and the `usage ()` function. The `Cmdline ()` function is where the meat of Genparse processing occurs. It takes as arguments a pointer to an `arg_t` struct which will be filled with the values of the options, and `argc` and `argv` which should be passed as the `main ()` function receives them. Genparse asumes that `arg_t` is a valid pointer to an `arg_t` struct, the calling program is responsible for properly allocating memory for it. Typically, the parsing function should be called at the beginning of the program.

The usage function lists the command line options for the program, as well as any mandatory command line parameters. Once this information is displayed, the program is terminated. For example, the usage function output for `mycopy3`, as invoked by the '`-h`' option, is as follows:

```
usage: mycopy3 [ -iohqv ] file
  [ -i ] [ --iterations  ] Number of times to output <file>.  (default = 1)
  [ -o ] [ --outfile  ] Output file.
  [ -h ] [ --help  ] Display help information.  (default = 0)
  [ -v ] [ --version  ] Output version.  (default = 0)
```

After displaying a brief list of all single-character options and mandatory options, the usage message lists all options in short and long forms, along with user-defined descriptions and each option's default value.

2.3.2 Parser Files

In this section we examine the parser file generated from running Genparse on `mycopy3.gp`.

```
/* mycopy3_clp.c */

#include <string.h>
#include <stdlib.h>
#include <getopt.h>
```

[2] The name of the command line parsing function can be user-defined. See Chapter 3 [Genparse Options], page 29.

```
#include "mycopy3_clp.h"

static struct option const long_options[] =
{
  {"iterations", required_argument, NULL, 'i'},
  {"outfile", required_argument, NULL, 'o'},
  {"help", no_argument, NULL, 'h'},
  {"version", no_argument, NULL, 'v'},
  {NULL, 0, NULL, 0}
};

/*----------------------------------------------------------------------------
**
** Cmdline ()
**
** Parse the argv array of command line parameters
**
**--------------------------------------------------------------------------*/

void Cmdline (struct arg_t *my_args, int argc, char *argv[])
{
  extern char *optarg;
  extern int optind;
  int c;
  int errflg = 0;

  my_args->i = 1;
  my_args->o = NULL;
  my_args->h = false;
  my_args->v = false;

  optind = 0;
  while ((c = getopt_long (argc, argv, "i:o:hv", long_options, &optind)) != EOF)
    {
      switch (c)
        {
        case 'i':
          my_args->i = atoi (optarg);
          if (my_args->i < 0)
            {
              fprintf (stderr, "parameter range error: i must be >= 0\n");
              errflg++;
            }
          break;

        case 'o':
          my_args->o = optarg;
```

```
              break;

            case ’h’:
              my_args->h = true;
              usage (EXIT_SUCCESS, argv[0]);
              break;

            case ’v’:
              my_args->v = true;
              break;

            default:
              usage (EXIT_FAILURE, argv[0]);

            }
        } /* while */

      if (errflg)
        usage (EXIT_FAILURE, argv[0]);

      if (optind >= argc)
        my_args->optind = 0;
      else
        my_args->optind = optind;
    }

    /*----------------------------------------------------------------------------
    **
    ** usage ()
    **
    ** Print out usage information, then exit
    **
    **--------------------------------------------------------------------------*/

    void usage (int status, char *program_name)
    {
      if (status != EXIT_SUCCESS)
        fprintf (stderr, "Try ‘%s --help’ for more information.\n",
                program_name);
      else
        {
          printf ("\
    usage: %s [ -iohv ] file\n\
    Print a file for a number of times to stdout.\n\
       [ -i ] [ --iterations ] (type=INTEGER, range=0..., default=1)\n\
              Number of times to output <file>.\n\
              File should be text format!\n\
       [ -o ] [ --outfile ] (type=STRING)\n\
              Output file name.\n\
       [ -h ] [ --help ] (type=FLAG)\n\
              Display this help and exit.\n\
       [ -v ] [ --version ] (type=FLAG)\n\
              Output version information and exit.\n", program_name);
        }
      exit (status);
    }
```

The parser file consists of two main functions. The `usage ()` function displays the program's usage information, then terminates. The parsing function, named `Cmdline ()` in this case[3], reads the command line and fills an `arg_t` struct with the command line parameters.

Since the `usage ()` function is straightforward, we will not examine it in detail. Instead, we will focus our attention on `Cmdline ()`. This function begins by defining a `struct long_options` array based on the specification in the `mycopy3.gp` file. This array will tell `getopt_long ()` what options to expect on the command line. The `struct arg_t` is then initialized and the default parameter values from `mycopy3.gp` are set. Once this is complete, `getopt_long ()` is looped through until all command line options have been processed. Each option has its own `case` in the `switch` statement. While this processing is fairly simple, there are several options worth examining in more detail.

- For the '`-h`' option, the `usage ()` function is automatically called.
- Range checking for '`-i`' occurs. If '`-i`' has a negative value, an error message is printed to *stderr* and the error flag is raised.

After the loop over `getopt_long ()` is complete, the error flag is checked. If it is raised, or if the help option has been set, the `usage ()` function is called.

At the end of `Cmdline ()`, the behavior of `optind` is modified slightly. While `optind` is returned to the caller in the `struct arg_t`, we set it to 0 if there are no non-option command line parameters. Otherwise, we pass it back as is, so that it can be used as a pointer into `argv`.

2.3.3 Main Program

Next to specification of the Genparse file, the most important part of using Genparse is interfacing it with user code. In this section, we show the main program code for `mycopy3.c` and describe how the routines created by Genparse are used.

```
/* mycopy3.c */

#include <stdio.h>
#include <stdlib.h>
#include "mycopy3_clp.h"

#define VERSION "3.0"
```

[3] The name of the parser function can be set by the user. See Chapter 3 [Genparse Options], page 29.

```
int main (int argc, char *argv[])
{
  int c, i;
  FILE *fp, *ofp;
  struct arg_t a;

  Cmdline (&a, argc, argv);

  if (a.v)
    {
      printf ("%s version %s\n", argv[0], VERSION);
      exit (0);
    }

  if (a.o)
    ofp = fopen (a.o, "w");
  else
    ofp = stdout;

  if (!a.optind)
    fp = stdin;
  else
    fp = fopen (argv[a.optind],"r");

  for (i = 0; i < a.i; i++)
    {
      while ((c = fgetc (fp)) != EOF)
fputc (c, ofp);
      rewind (fp);
    }

  fclose (fp);
  return 0;
}
```

The `mycopy3.c` module begins by including `mycopy3_clp.h`, which is necessary for the definition of the `arg_t` struct and the parser function prototypes. The `Cmdline ()` function fills an `arg_t` struct with the command line options. While the parsing function does not always have to be called before all other processing, it must be called before any command line parameters are used.

The program then checks a number of the parameters, as returned in the structure. In particular, if the '`-v`' option is set, the version number is displayed and then the program is terminated.

If the `optind` pointer is set to 0, indicating that there are no non-option parameters on the command line, the input is redirected to *stdin*, If the output file is not specified with the '`-o`' option, output is redirected to *stdout*.

2.4 Include Files and Callbacks

So far, our examples have only considered the basic features of Genparse. In fact, these features are probably more than enough for 90% of all command line parsing needs. However, some programs require additional flexibility. In this section, we explore some of the advanced features of Genparse.

Include files may be specified at the beginning of a Genparse file. They are listed in C style, e.g., `#include <file.h>` or `#include "file.h"`. One of the possible uses of include files are for when a macro needs to be used in a Genparse file. Consider the following modification to `mycopy3.gp`.

```
/* mycopy3.gp */
#include <mycopy3.h>
i / iterations int 1 [0...MAX] "Number of times to output <file>."
o / outfile string {""} "Output file."

#usage_begin
usage: __PROGRAM_NAME__ __OPTIONS_SHORT__ file
Print a file for a number of times to stdout.
__GLOSSARY__
#usage_end
```

This example assumes that the macro `MAX` is defined in `mycopy3.h`.

In order to demonstrate the utility of callback functions, let's further modify `mycopy3.gp`. In fact, let's just call it `mycopy4.gp`.

```
/* mycopy4.gp */
#include <mycopy4.h>
my_callback ()
i / iterations int 1 [1..MAX] "Number of times to output <file>."
o / outfile string {""} outfile_cb () "Output file."

#usage_begin
usage: __PROGRAM_NAME__ __OPTIONS_SHORT__ file
Print a file for a number of times to stdout.
__GLOSSARY__
#usage_end
```

This file instructs Genparse to create a global callback function `my_callback ()` and the option callback function `outfile_cb ()`.

For these callbacks, Genparse adds prototypes to the header file, a call to the parser file, and callback skeletons in a callback file. Rather than display the whole header and parse files, we'll just show the lines that are added.

Callback functions are useful if extra processing needs to occur before one or more parameters are used by the main program. For example, if one parameter is dependent on the values of two others, a user-defined global callback function can contain the logic to check for the proper conditions.

```
/* mycopy4_clp.h */
/* global and local callbacks */
int my_callback (struct arg_t *);
int outfile_cb (char *);
```

In the mycopy4_clp.h file, prototypes for the global and the option callback functions are added.

```
/* mycopy4_clp.c */
  case 'o':
    my_args->o = optarg;
    if (!outfile_cb (my_args->o))
      usage (EXIT_FAILURE, argv[0]);
    break;

...

if (!my_callback (my_args))
  usage (argv[0]);
```

In the mycopy4_clp.c file, a call to the callback functions is made. If a 0 is returned, an error is assumed and the usage function is called.

The main difference in Genparse behavior that including callbacks produces is the creation of a callback file. This file contains skeletons of all of the callback functions. It is up to the user to fill them in. Note that callbacks should return 0 on error and non-zero otherwise.

```
/* mycopy4_clp_cb.c */

#include <stdio.h>
#include "mycopy4_clp.h"

/*----------------------------------------------------------------------------
**
** my_callback ()
**
** User defined global callback.
**
**--------------------------------------------------------------------------*/

int my_callback (struct arg_t *a)
{
  return 1;
}

/*----------------------------------------------------------------------------
**
** outfile_cb ()
**
** User defined parameter callback.
```

```
**
**--------------------------------------------------------------------------*/

int outfile_cb (char * var)
{
  return 1;
}
```

Note that the `struct arg_t` structure must be passed to the global callback, while the option value is expected to be passed in character array format to option callbacks.

2.5 C++ Output

All of our example so far have shown Genparse creating C code. Genparse also supports C++ output. This section examines the difference between C and C++ output and how to interface a program with a command line parsing class created by Genparse.

As with C output, Genparse creates two or three C++ output files: a header file, a parser class file, and a callback file. We'll use `mycopy4.gp` as input to create these files. We invoke Genparse as follows.

```
genparse -l cpp -o mycopy4_clp mycopy4.gp
```

The three created files are named, `mycopy4_clp.h`, `mycopy4_clp.cc`, and `mycopy4_clp_cb.cc`. We'll walk through each one in turn.

2.5.1 Header File

The code for `mycopy4_clp.h` appears below.

```
/* mycopy4_clp.h */

#ifndef CMDLINE_H
#define CMDLINE_H

#include <iostream>
#include <string>
#include "mycopy4.h"

/*----------------------------------------------------------------------------
**
** class Cmdline
**
** command line parser class
**
**--------------------------------------------------------------------------*/

class Cmdline
{
private:
  /* parameters */
  int _i;
```

```
    std::string _o;
    bool _h;
    bool _v;

    /* other stuff to keep track of */
    std::string _program_name;
    int _optind;

  public:
    /* constructor and destructor */
    Cmdline (int, char **) throw (std::string);
    ~Cmdline (){}

    /* usage function */
    void usage (int status);

    /* return next (non-option) parameter */
    int next_param () { return _optind; }

    /* callback functions */
    bool my_callback ();
    bool outfile_cb ();

    int i () { return _i; }
    std::string o () { return _o; }
    bool h () { return _h; }
    bool v () { return _v; }
  };

  #endif
```

The header file contains the definition of the command line parser class. The class defines a logical structure that puts different requirements on the main program than when the output code is in C. We summarize the differences between C and C++ output below.

- Each parameter value is stored in a private member variable, and must be accessed through a call to the member function named with the short form of the option. If Genparse was called with the option '`--longmembers`' then this function is named by the long form of the option.
- The name of the executable is stored in a private member variable, but is not available to the via the interface (the main program has access to `argv[0]`).
- A copy of the `optind` variable is stored in a private member variable, and is accessible through the `next_param ()` member function.
- All callbacks are public member functions.
- The usage function is a public member function.
- The constructor throws a string exception if command line parsing fails.

2.5.2 Parser File

The parser file defines the non-inlined member functions; i.e., the constructor and the usage function. The code for `mycopy4_clp.cc` appears below.

```
/* mycopy4_clp.cc */

#include <getopt.h>
#include <stdlib.h>
#include "mycopy4_clp.h"

/*----------------------------------------------------------------------------
**
** Cmdline::Cmdline ()
**
** Constructor method.
**
**--------------------------------------------------------------------------*/

Cmdline::Cmdline (int argc, char *argv[]) throw (std::string )
{
  extern char *optarg;
  extern int optind;
  int c;

  static struct option long_options[] =
  {
    {"iterations", required_argument, NULL, 'i'},
    {"outfile", required_argument, NULL, 'o'},
    {"help", no_argument, NULL, 'h'},
    {"version", no_argument, NULL, 'v'},
    {NULL, 0, NULL, 0}
  };

  _program_name += argv[0];

  /* default values */
  _i = 1;
  _h = false;
  _v = false;

  optind = 0;
  while ((c = getopt_long (argc, argv, "i:o:hv", long_options, &optind)) != EOF)
    {
      switch (c)
        {
        case 'i':
          _i = atoi (optarg);
          if (_i < 1)
            {
              std::string s;
              s += "parameter range error: i must be >= 1";
              throw (s);
            }
          if (_i > MAX)
            {
              std::string s;
              s += "parameter range error: i must be <= MAX";
              throw (s);
            }
          break;

        case 'o':
          _o = optarg;
          if (!outfile_cb ())
            this->usage (EXIT_FAILURE);
          break;

        case 'h':
          _h = true;
          this->usage (EXIT_SUCCESS);
          break;

        case 'v':
          _v = true;
          break;

        default:
          this->usage (EXIT_FAILURE);

        }
    } /* while */

  _optind = optind;
  if (!my_callback ())
    usage (EXIT_FAILURE);

}

/*----------------------------------------------------------------------------
**
** Cmdline::usage ()
**
** Print out usage information, then exit.
**
```

```
**-------------------------------------------------------------------------*/

void Cmdline::usage (int status)
{
  if (status != EXIT_SUCCESS)
    std::cerr << "Try `" << _program_name << " --help' for more information.\n";
  else
    {
      std::cout << "\
usage: " << _program_name << " [ -iohv ] file\n\
Print a file for a number of times to stdout.\n\
   [ -i ] [ --iterations ] (type=INTEGER, range=1...MAX, default=1)\n\
          Number of times to output <file>.\n\
          do it like this\n\
   [ -o ] [ --outfile ] (type=STRING)\n\
          Output file.\n\
   [ -h ] [ --help ] (type=FLAG)\n\
          Display this help and exit.\n\
   [ -v ] [ --version ] (type=FLAG)\n\
          Output version information and exit.\n";
    }
  exit (status);
}
```

As can be seen from this example, the C++ parser is very similar to the C parser we discussed in Section 2.3.2 [Parser Files], page 8. The main differences are that all strings are stored in C++ `string` format[4].

It is important to note that the callback file is actually included into the parser file. This is because callbacks are implemented as member functions of the parser class, and most C++ compilers will not allow splitting a class's member functions across more than one file. The bottom line of all this is that you don't have to link in the callback file, just the parser file.

2.5.3 Callback File

The callback file defines skeleton callback routines for the user to fill in. Their syntax is virtually identical to the C output case, except that their return values are `bool` rather than `int`. The code for `mycopy4_clp_cb.cc` appears below.

```
/* mycopy4_clp_cb.cc */

#include "mycopy4_clp.h"

/*--------------------------------------------------------------------------
**
** Cmdline::my_callback ()
**
```

[4] Older C++ compilers may not support built-in strings. If this is a problem, upgrade your compiler! It's too old anyway.

```
** Global callback.
**
**----------------------------------------------------------------------------*/

bool Cmdline::my_callback ()
{
  return true;
}

/*----------------------------------------------------------------------------
**
** Cmdline::outfile_cb ()
**
** Parameter callback.
**
**----------------------------------------------------------------------------*/

bool Cmdline::outfile_cb ()
{
  return true;
}
```

2.5.4 Main Program

Due to the syntactic differences between C and C++, the C++ main program must interact with the command line parser class in a different fashion than in the C case. The code for `mycopy4.cc` appears below.

```
/* mycopy4.cc */

#include <cstdlib>
#include <iostream>
#include <fstream>
#include "mycopy4_clp.h"

using namespace std;

#define VERSION "3.0"

int main (int argc, char *argv[])
{
  int i;
  char c;
  ifstream input_file;
  ofstream output_file;
  bool ofile = false, ifile = false;

  Cmdline cl (argc, argv);
  if (cl.v ())
    {
      cout << argv[0] << " version " << VERSION << endl;
      exit (0);
    }

  if (!cl.o ().empty ())
    {
      output_file.open (cl.o ().c_str ());
      ofile = true;
    }

  if (cl.next_param ())
    {
      input_file.open (argv[cl.next_param ()]);
      ifile = true;
    }

  for (i = 0; i < cl.i (); i++)
    {
      if (ifile) c = input_file.get ();
      else cin >> c;

      while (c != EOF)
{
  if (ofile) output_file.put (c);
  else cout << c;

  if (ifile) c = input_file.get ();
  else cin >> c;
}
      if (ifile)
{
  input_file.clear ();
  input_file.seekg (0);
}
    }

  input_file.close ();
  output_file.close ();

  return 0;
}
```

Although mycopy4 provide almost identical output and functionality as that of mycopy3, it must access all command line parameters and related information through the command line parser class interface.

2.6 Java Output

This section shows the Java output and how to interface a Java program with a command line parsing class created by Genparse.

For Java, Genparse creates a Java interface, a parser class that implements it and an exception class that may be thrown by the parser. A separate output file is written for each of them. We'll use mycopy5.gp as input to create the two output files. Genparse generates Java code as follows.

```
genparse -l java -o Cmdline mycopy5.gp
```

The three created files are named, CmdlineInterface.java, Cmdline.java and CmdlineEx.java. We'll walk through each one in turn.

2.6.1 Java Interface

```
/* CmdlineInterface.java */

/*----------------------------------------------------------------------------
**
** interface CmdlineInterface
**
** Interface of the command line parser class
**
**--------------------------------------------------------------------------*/

public interface CmdlineInterface
{
  /* usage function */
  void usage (int status, String program_name);

  /* return next (non-option) parameter */
  int next_param ();

  /* callback functions */
  boolean my_callback ();
  boolean outfile_cb ();

  /* getter functions for command line parameters */
  int i ();
  String o ();
  boolean h ();
  boolean v ();
};
```

Each parameter value is stored in a private member variable, and must be accessed through a call to the member function named with the short form of the option. If Genparse was called with the option '`--longmembers`' then this function is named by the long form of the option.

A copy of the `optind` variable is stored in a private member variable, and is accessible through the `next_param ()` member function.

2.6.2 Java Implementation

```
/* Cmdline.java */

import gnu.getopt.LongOpt;
import gnu.getopt.Getopt;

/*----------------------------------------------------------------------------
**
** class Cmdline ()
**
** Command line parser class.
**
**--------------------------------------------------------------------------*/

public class Cmdline implements CmdlineInterface
{
  /* parameters */
  private int _i;
  private String _o;
  private boolean _h;
  private boolean _v;

  /* Name of the calling program */
  private String _executable;

  /* next (non-option) parameter */
  private int _optind;

  /* Must be constructed with parameters. */
  public Cmdline (String[] argv) throws CmdlineEx
  {
    /* character returned by optind () */
    int c;

    LongOpt[] longopts = new LongOpt[4];
    longopts[0] = new LongOpt ("iterations", LongOpt.REQUIRED_ARGUMENT, null, 'i');
    longopts[1] = new LongOpt ("outfile", LongOpt.REQUIRED_ARGUMENT, null, 'o');
    longopts[2] = new LongOpt ("help", LongOpt.NO_ARGUMENT, null, 'h');
    longopts[3] = new LongOpt ("version", LongOpt.NO_ARGUMENT, null, 'v');
    _executable = Cmdline.class.getName ();

    /* default values */
    _i = 1;
    _h = false;
    _v = false;

    Getopt g = new Getopt (_executable, argv, "i:o:hv", longopts);
    while ((c = g.getopt ()) != -1)
    {
      switch (c)
      {
        case 'i':
          _i = Integer.parseInt (g.getOptarg ());
          if (_i < 1)
            throw new CmdlineEx ("parameter range error: i must be >= 1");
          if (_i > 10)
            throw new CmdlineEx ("parameter range error: i must be <= 10");
          break;

        case 'o':
          _o = g.getOptarg ();
          if (!outfile_cb ())
            usage (-1, _executable);
          break;

        case 'h':
          _h = true;
          usage (0, _executable);
          break;

        case 'v':
          _v = true;
          break;

        default:
          usage (-1, _executable);

        }
    } /* while */

    _optind = g.getOptind ();
    if (!my_callback ())
      usage (-1, _executable);

  }
```

```
    public void usage (int status, String program_name)
    {
      if (status != 0)
        {
          System.err.println ("Try `" + program_name + " --help' for more information.");
        }
      else
        {
          System.out.println (
  "usage: " + program_name + " [ -iohv ] file\n" +
  "Print a file for a number of times to stdout.\n" +
  "    [ -i ] [ --iterations ] (type=INTEGER, range=1...10, default=1)\n" +
  "           Number of times to output <file>.\n" +
  "           do it like this\n" +
  "    [ -o ] [ --outfile ] (type=STRING)\n" +
  "           Output file.\n" +
  "    [ -h ] [ --help ] (type=FLAG)\n" +
  "           Display this help and exit.\n" +
  "    [ -v ] [ --version ] (type=FLAG)\n" +
  "           Output version information and exit.");
        }
      System.exit (status);
    }

    /* return next (non-option) parameter */
    public int next_param () { return _optind; }

    /* Callback functions */
    /* Derive your own class and overwrite any of the callback  */
    /* functions if you need customized callbacks. */
    public boolean my_callback () { return true; }
    public boolean outfile_cb () { return true; }

    /* getter functions for command line parameters */
    public int i () { return _i; }
    public String o () { return _o; }
    public boolean h () { return _h; }
    public boolean v () { return _v; }
  }
```

There is no default constructor. Instead the parser class must be constructed with an array of strings which is usually the argument to the `main ()` function.

The constructor throws an exception if command line parsing fails. The exception type is defined in See Section 2.6.3 [Java Exception Class], page 26.

Note that no separate callback file is generated like for C or C++. The parser class contains default implementations for each of the callback functions, if you need customized callbacks then simply derive your own class from the generated parser class.

2.6.3 Java Exception Class

```
/* CmdlineEx.java */

/*----------------------------------------------------------------------------
**
** class CmdlineEx
**
** Exception class thrown by the command line parser class
**
**--------------------------------------------------------------------------*/

public class CmdlineEx extends RuntimeException
{
  public CmdlineEx (String text) { super (text); }
}
```

This exception class is always the same of course. Only the name of the class would change if –parsefunc was set with a different name than Cmdline.

2.6.4 Main Java Program

The example code for Java is very similar to the C++ example (See Section 2.5.4 [Main C++ Program], page 20.).

```
/* mycopy5.java */

import gnu.getopt.LongOpt;
import gnu.getopt.Getopt;
import java.io.*;

class mycopy5
{
  public static void main (String args[])
  {
    final String _executable = "mycopy5";   /* Don't know how to set this automaticall
    final String VERSION = "3.0";
    final int EOF = - 1;  /* Is EOF predefined in Java? */

    int i;
    int c;
    int length;
    OutputStreamWriter output_file;
    InputStreamReader input_file;
    boolean ofile = false, ifile = false;
```

```
Cmdline cl = new Cmdline (args);

if (cl.v ())
{
  System.out.println (_executable + " version " + VERSION);
  System.exit (0);
}

try
{
  length = cl.o ().length ();
  if (length != 0)
  {
    output_file = new OutputStreamWriter (new FileOutputStream (cl.o ()));
    ofile = true;
  }
  else
  {
    output_file = new OutputStreamWriter (System.out);
  }

  if (cl.next_param () != 0)
  {
    input_file = new InputStreamReader (new FileInputStream (args[cl.next_param ()
    ifile = true;
  }
  else
  {
    input_file = new InputStreamReader (System.in);
  }

  for (i = 0; i < cl.i (); i++)
  {
    c = input_file.read ();

    while (c != EOF)
    {
      output_file.write (c);
      c = input_file.read ();
    }
    if (ifile)
    {
      input_file.close ();
      input_file = new InputStreamReader (new FileInputStream (args[cl.next_param
    }
  }
```

```
          input_file.close ();
          output_file.close ();
        }
        catch (IOException ex)
        {
          System.out.println ("File I/O error: " + ex);
        }

        System.exit (0);
      }
    }
```

2.7 Idiosyncrasies

Although Genparse has been designed to be as flexible as possible, it will always be more limited than manual command line parsing or using `getopt ()`. In this section, we document some of the strange and unusual behavior of Genparse.

- In order to specify an option that only has a long form, you must specify the token "NONE" in place of the short option. For example

  ```
  NONE / longonly   flag
  ```

  specified a flag option that only has a long form ('`--longonly`').
- The '`-h`' and '`-v`' options will always exist. Their default behavior can be overridden, but Genparse cannot create parsers without them. (See Chapter 3 [Genparse Options], page 29.)
- In a parser, the '`-h`' option will automatically call the usage function if and only if the long form of the option is '`--help`'. Otherwise, Genparse assumes that the user has overridden '`-h`' and treats it like any other option.
- For C output you must link in both the parser and the callback files. For C++ output, you only need to link in the parser file. See Section 2.5.2 [Parser File], page 17.

3 Genparse Options

In this section we present the command line options that Genparse accepts, along with their default values. Genparse's command line parsing functions were created by Genparse (talk about bootstrapping), so you can expect Genparse to behave like any other program with Genparse-enabled command line parsing.

- '`-c`' / '`--cppext`': C++ file extension. There are a number of valid C++ file extensions, such as "C", "cc", "cpp",and "cxx". This option allows the user to specify which one should be used for the C++ files created by Genparse. The default value is "cc". If C++ is not the output language, this option is ignored.
- '`-d`': Debug mode. Setting this flag turns on debugging in the form of logging to a file and `bison` debug output to *stderr*. The name of the debug/log file can be specified with the '`-f`' option (see below). The default value is off. Only useful for debugging Genparse itself.
- '`-f`' / '`--logfile`': Name of log file. Only used if debugging ('`-d`' option) is turned on. All debug output will be written to this file. Currently this consists of the processing of the Genparse file and dumping the state of the command line parameter list class. The default value is "genparse.log". Only useful for debugging Genparse itself.
- '`-g`' / '`--gnulib`': Use GNU Compatibility Library (Gnulib, see `http://www.gnu.org/software/gnulib/` Only available for C output. Allows some more types (unsigned long, intmax_t etc.) for which Gnulib provides conversion functions.
- '`-h`' / '`--help`': Help instructions. This option displays the usage of Genparse with a list of all command line options, then terminates the program. The default is off.
- '`-i`' / '`--internationalize`': Put internationalization macro _() around text output so that the generated program can be internationalized using the GNU gettext command. Presently only implemented for C output.
- '`-l`' / '`--language`': Output language. The programming language in which Genparse writes the output files. Currently only C, C++ and Java are supported. The default is C. To indicate C++, the following strings may be used: "c++", "cpp", "cc", and "cxx". In order to generate Java code use "java" or "Java".
- '`-m`' / '`--longmembers`': Use long options for the members of the parser class (struct). The default is to use the short representation except if there is only a long representation defined in the genparse file. If this option is set then the behavior is reverted. The long representation is used then except if there is only a short representation defined.
- '`-o`' / '`--outfile`': Output file name. Specifies the main part of the output file name. The extension will be determined by the output language and possibly by other options. For example, when the output language is C, giving this option an argument of "file" will result in output file names of "file.h", "file.c" and "file_cb.c" for the header, parser, and callback files, respectively. Default value is "parse_cl". Use the '`-D`' option in order to specify a directory for the results.
- '`-p`' / '`--parsefunc`': Name of the parsing function/class. This option allows the user to specify the name of the function (for C) or class (for C++ and Java) that does the actual command line parsing. Default value is "Cmdline". In Java this value is also used as a prefix for the names of the interface and the exception class. Leaving the default names the interface "CmdlineInterface" and the exception class "CmdlineEx".

- '`-P`' / '`--manyprints`': Output help text for every command line parameter in a separate print command.
- '`-q`' / '`--quiet`': Quiet mode.
- '`-s`' / '`--static-headers`': Keep the descriptive header on top of the generated files static. Without this option genparse prints creation date and time, Linux kernel version, kernel build time, computer architecture name, host name and user name.
- '`-v`' / '`--version`': Version. If this option is set, the version number is displayed, then the program is terminated. The default value is off.
- '`-D`' / '`--directory`': Directory for storing results.

The '`-h`' and '`-v`' options can be overridden by defining them to be something else in the Genparse file. However, they cannot be turned off completely. In other words, you can define your own '`-h`' and '`-v`' options or let Genparse create them with the default behavior.

4 Genparse File Syntax

4.1 Global Definitions

The following definitions are allowed on top of the parameter definitions. All of them are optional. They may appear in any order.

- **Include files**: Include statements in the C syntax, e.g. `#include "file.h"` or `#include <file.h>`. Ignored in languages other than C and C++. Only 1 include statement per line.
- **Mandatory parameters**: Parameters which must be specified. They are not bound to any command line options. Example: `#mandatory x`. Use multiple `#mandatory` statements in order to specify multiple mandatory parameters. Only 1 `#mandatory` statement per line. Note that Genparse does not check for mandatory parameters, they are only printed in the `usage ()` function with the `__MANDATORIES__` directive (See [__MANDATORIES__], page 34.). **Deprecated: add mandatory parameters in the #usage section instead.**
- **Exit value**: Defines the exit value in case of an error. Default is `EXIT_FAILURE`. Example: `#exit_value -1` or `#exit_value MY_FAILURE`.
- **#break_lines**: Width to which lines shall be broken on the help screen. If no `#break_lines` directive is specified then lines will be printed exactly as given in the genparse file. Note that genparse doesn't know the width of macros included from other files, so automatic line breaking will probably not work properly if for example `__STRING__` macros (See [__STRING__], page 34.) are used. Lines can still be broken manually using the `__NL__` macro (See [__NL__], page 34.) in places where Genparse doesn't break lines as expected.
- **#no_struct**: If `#no_struct` is specified then no struct will be defined which will be filled with the command line parameters in the generated parser. This may be useful if you want to add your own code with `__CODE__` statements instead (See [__CODE__], page 33.) Only supported for C output.
- **#export_long_options**: If `#export_long_options` is defined then a function `get_long_options ()` is added which exports the longoptions array used by `getopt_long ()`. This directive is only available for C output, for other languages it is ignored.
- **Global callback**: The name of a global callback function which will be called after all parameter specific callback functions. Example: `my_callback ()`. Only 1 global callback function is allowed.

4.2 Parameter Definitions

Each command line parameter must be defined in the form

```
short_names[*|!] [/ long_name[*|!][=opt_name]] type [ options ]
```

A short_name is a single letter (small or capital) or a single digit. long_name is a longer (more descriptive) option name. On the command line a short name will be preceded by a single dash (e.g. `-a`) and a long version will be preceded by two dashes (e.g. `--all`). If a long parameter name is not necessary, you may specify only the short one (and the slash need not appear). In order to specify a parameter that only has a long name set short_names to `NONE`. It is possible to have multiple short options, so for example setting short_name to 'aA' and long_name to 'all' would allow to specify the command line switch as `-a` or `-A` or `--all`, all of them doing the same thing. long options can be followed by a descriptive designation (See [opt_name], page 34.).

A `*` after `short_name` or `long_name` makes the argument optional. This can be specified for short and long options separately.

A `!` after `short_name` or `long_name` makes the option boolean. This allows one to combine a boolean short option with a long option with an optional or mandatory argument or to combine a boolean long option with a short option with an optional or mandatory argument. A `!` doesn't make sense if the option's type is `flag`. Examples:

```
o* / oparam* string "Both short and long option have an optional"
"argument"
p* / pparam string "Short option has an optional argument,"
"long option requires an argument."
q / qparam* string "Short option reqires an argument,"
"long option has an optional argument."
P* / Pparam!  string "Short option has an optional argument,"
"long option none."
Q!/ Qparam* string "Short option has no argument, long option has an"
"optional argument."
```

type must be one of `int float char string` or `flag`. The first four should be self-explanatory. The last is a "switch" option that takes no arguments. For C output and if '`--gnulib`' (See [gnulib], page 29.) is set on the command line additionally the following types are allowed: `long` (for long int), `ulong` (for unsigned long int), `intmax` (for intmax_t, defined in Gnulib), `uintmax` (for uintmax_t), `double`.

The following four options are supported. They may appear in any order and except for descriptions only one of each field may be defined per option.

- A **default value** for the parameter. For a string this is just the plain default value, whatever it is. For strings, a default must be specified within braces and quotes, and may include whitespace, e.g. `{"`my default value`"}`. For a char parameter it must be enclosed in single quotes, e.g. 'a' or '\n'.
- A **range** of values within brackets. The low and high values are specified between a range specifier (either '...' or '..'). Either the high or the low value may be omitted for a range bounded on one side only. The parameter will be checked to make sure that it lies within this range.
- A **callback function**. This function is called after any range checking is performed. The purpose of the callback to do validity checking that is more complicated than can be specified in the genparse file. For example, you might write a program that requires input to be prime numbers, strings of a certain length, etc.
- A **description** in double quotes. It is printed by the `usage ()` function. If one line is not enough then specify multiple descriptions, one per line and each of them in double quotes. If the description starts in the 1st column in the Genparse file then it will also be printed in the 1st column in the `usage ()` function.

- A **Genparse include file**. Includes another Genparse (.gp) file, e.g `#gp_include another.gp`. Only parameter definitions are allowed in the included file, no global directives.
- An **__ERR_MSG__(err_txt)** directive. Specifies the error message which is printed when the argument could not be converted. Example: `__ERR_MSG__("%s: invalid argument")`. This message will be printed when either the conversion function failed or when the argument was out of range (See [range], page 32.). Assumes to contain one `%s` which will be replaced with the agrument which could not be converted. Only available when Genparse is invoked with '`--gnulib`' (See [gnulib], page 29.), ignored otherwise.

  Optionally a conversion function can be added as a second argument, e. g. `__ERR_MSG__("%s: invalid argument", quotearg)`. This would lead to an error message like `error (EXIT_FAILURE, 0, "%s: invalid argument", quotearg (optind))`.
- An **__ADD_FLAG__** directive. Makes sense only if the command line parameter is not already a flag, in this case an additional flag parameter is added which will be set if the command line parameter was specified on the command line. This option is automatically set if a parameter has an optional argument.
- A **__CODE__(statements)** directive. The specified code statements are copied literally. Example: `__CODE__(printf ("Parameter x was set");)`. The specified code can extend over more than one line. In order to give Genparse the chance to indent the code properly, do not mix space and tab indentations in one `__CODE__` statement.
- A **__STORE_LONGINDEX__** directive. Instructs Genparse to add an interer type field to the result class which will be set to the longindex variable (last argument in the call to `getopt_long ()`). This new field will get the same name as the result field it is related to but with an `_li` postfix.

4.3 Usage Function

Genparse also generates a usage () function which prints a help text to stdout about the usage of the program for which Genparse is generating the parser. It is automatically executed when

- The command line option -h is set.
- An invalid command line option is given.
- The optind_long function call (which is the heart of every Genparse generated parser) returns an error code.

This usage function can be customized by specifying a usage section at the bottom of the Genparse file. If no such section is specified it defaults to

```
#usage_begin
usage: __PROGRAM_NAME__ __OPTIONS_SHORT__ __MANDATORIES__
__GLOSSARY__
#usage_end
```

The usage section starts with `#usage_begin` and ends with `#usage_end`. Any text between is printed verbatim except for the following keywords, which will be replaced as listed below:

- `__PROGRAM_NAME__`: The program name. In C and C++ the program name is given in `argv[0]`.
- `__OPTIONS_SHORT__`: A list of available short form options, e.g. `[ -abc ]`.
- `__MANDATORIES__`: A list of all mandatory parameters as defined with `#mandatory` commands (See Section 4.1 [Global Definitions], page 31.). **Deprecated: List mandatory parameters here directly.**
- `__GLOSSARY__`: A description of all command line options. This is the information given for the parameter definitons (see See Section 4.2 [Parameter Definitions], page 31.) in human readable form. It includes the parameter type, default, range and any comments. A line which contains `__GLOSSARY__` is replaced by the glossary of the parameters, any other text in the same line is ignored. Example:

  ```
  [ -h ] [ --help ] (type=FLAG)
         Display help information.
  ```
- `__GLOSSARY_GNU__`: Same as `__GLOSSARY__` but in GNU style. Optionally followed by an integer in brackets which specifies the indentation of the descriptive text (e.g. `__GLOSSARY__(30)`). Default indentation is 24. Example:

  ```
  -h, --help            Display help information.
  ```
- `__STRING__(s)`: A string constant, in C probably a string macro defined with the #define preprocessor command. This macro can be imported from another file using the include directive in the genparse file (See [Include files], page 31.). Ignored when generating Java output.
- `__INT__(x)`: An integer constant, in C probably an integer macro defined with the #define preprocessor command. This macro can be imported from another file using the include directive in the genparse file (See [Include files], page 31.). Ignored when generating Java output.
- `__CODE__(statements)`: See [__CODE__], page 33.
- `__DO_NOT_DOCUMENT__`: Any line which contains this macro will not be printed in the `usage ()` function. Can be used for implementing command line parameters without listing them on the help screen.
- `__NL__`: New line. Useful for breaking lines manually while automatic line breaking is on (See [#break_lines], page 31.). Ignored when generating Java output.
- `__NEW_PRINT__`: Close the active print command and start a new one.
- `__COMMENT__(text)`: Comment in the code for printing the usage text.

long options can be followed by an `=` sign and an optional designation `opt_name` in the genparse file (See Section 4.2 [Parameter Definitions], page 31.) which can be referred to in the following description (See [parameter-description], page 32.). It will be used in the `usage ()` function only. For example the following genparse line

```
s / block-size=SIZE int "use SIZE-byte blocks"
```

will lead to the following line in the help screen

```
[ -s ] [ --block-size=SIZE ] (type=INTEGER)
       use SIZE-byte blocks
```

in genparse style (See [__GLOSSARY__], page 34.) or

```
-s, --block-size=SIZE   use SIZE-byte blocks
```

in GNU style (See [__GLOSSARY_GNU__], page 34.).

It is also possible to put square braces around the optional name in order to indicate that the argument is optional. This has no meaning for the generated parser however. Use `*` postfixes in order to make an argument optional (See [optional_arguments], page 32.).

```
s* / block*[=SIZE] int "use blocks."
                               "If SIZE is not given then they will get a size of 1kB
```

will lead to the following line in the help screen

```
-s, --block[=SIZE]   use blocks.
                     If SIZE is not given then they will get a size of 1kB.
```

For an example of the generated default usage () function in C See Section 2.3.2 [Parser Files], page 8.

4.4 Genparse File Grammar

In order to better understand the subtleties of the Genparse file format and its parsing, in the section we provide the formal grammar of Genparse files. This is a slightly edited version of the `bison` grammar. Items in capitals are tokens that are defined in the lex file.

```
all: globals entries usages

globals: /* empty */
        | globals global
        | global

global: include
        | mandatory
        | exit_value
        | break_lines
        | export_long_options
        | no_struct
        | global_callback

include: #include <FILENAME>

mandatory: #mandatory <FILENAME>

exit_value: #exit_value <VALUE>

break_lines: #break_lines <WIDTH>

export_long_options: #export_long_options

no_struct: #no_struct

global_callback: callback
```

```
entries: entries entry
        | entry

entry: new_entry
        | gp_include

new_entry: param type options

options: options option
        | option

option: default
        | range
        | callback
        | descs
        | store_longindex
        | err_msg
        | comment
        | code_lines

descs: descs desc
        | desc

err_msg: __ERR_MSG__ ( ALNUM )
        | __ERR_MSG__ ( QUOTED )

comment: __COMMENT__ ( COMMENT_STR )

gp_include: #gp_include <FILE>

param: short_params
        | NONE / long_param
        | short_params / long_param

short_params: VAR
        | VAR *
        | VAR !

long_param: multi_long_option
| multi_long_option mandatory_opt_name
| multi_long_option optional_opt_name

mandatory_opt_name: EQUAL C_VAR
| EQUAL OPT_NAME

optional_opt_name: [ mandatory_opt_name ]
```

```
multi_long_option: VAR
        | VAR *
        | VAR !

type: INT
        | LONG
        | ULONG
        | INTMAX
        | UINTMAX
        | FLOAT
        | DOUBLE
        | STRING
        | CHAR
        | FLAG

default: ALNUM
        | CHAR_VAL
        | QUOTED_STR

range: [ contiguous_range ]

contiguous_range: ALNUM range_spec more_range
        | range_spec more_range

more_range: ALNUM
        | C_VAR
        |

range_spec: ..
        | ...

callback: VAR ()

description: QUOTED_STR

usages: /* empty */
| usages usage
| usage

usage: USAGE_STR

code_lines: code_lines code_line
| code_line

code_line: single_code_line CODE_END
| CODE_END
;
```

```
single_code_line: CODE_LINE
| OPEN_ROUND_BRACE
| CLOSE_ROUND_BRACE
| single_code_line single_code_line
```

5 The Future

While there's practically no limit to the number of enhancements that could be made to Genparse, I'll limit this section to a number of features that actually have a chance of being implemented.

- Back up output files before overwriting them. Minimally, old callback files should be saved, because those are meant to be modified by the user. This isn't hard, but I'd like to do it `"right"` by encapsulating the functionality into a general-purpose file manipulation class.
- Enumerated types for strings. Currently command line strings are not checked for any sort of validity. This would allow the user to specify a list of valid values for a particular string. The syntax would probably be like an enumerated type in C.
- A way to distinguish between optional and mandatory non-option command line parameters.
- Support for more output languages. Perhaps tcl and perl first, and other scripting languages afterwards. This should be pretty easy to do since it would only require adding the appropriate member functions to the command line parameter list class and a little bit of supporting code here and there.
- Place `getopt.h`, `getopt.c`, and `getopt_internal.c` in a neutral shared directory so that others can copy them locally for their own use.

6 Some History

Long, long ago in a lab far, far away, I became frustrated with the time required to write command line parsing routines, even with `getopt ()`, for all of the new programs that I developed. I felt that it was tedious work that offered too many opportunities to cut corners and produce error-prone code. In late 1997, the seeds of Genparse fell together in my head, and I released version 0.1, written in C, on New Year's Day, 1998. The only output language supported was C, and it did not allow long options. Parameter types were limited to flags, integers, floats, and strings, and range checking was supported.

It soon became clear that although Genparse was fairly useful (I was already using it to create parsers for my own projects), it was severely limited and could use a number of additional features. With the help of a handful of people who provided feedback and constructive criticism, version 0.2 was soon released. New features included a more flexible Genparse file format (essentially, the current format) which was parsed by `lex` and `bison` rather than by my C code. Parameter descriptions and callback functions were now supported.

A few bug fixes later, version 0.2.2 was released. It was to remain current for over a year. In June 1999, I revisited Genparse to add the `#include` and `#mandatory` directives, as well as to clean up the code a bit and perform some minor bug stomping. The resulting version 0.3 was distributed with Debian/GNU Linux.

During my revision that produced 0.3, I became aware of the acute coding slop that had evolved. To call the output functions "spaghetti logic" would have been an understatement. Naturally, I needed to modularize Genparse, and the best way of doing so seemed to be a re-write in C++. In December 1999, I undertook this task. The code became separated into three logical sections:

- The parser, a `bison` grammar with supporting code.
- An internal representation, encapsulated in a C++ class.
- Output functions for each supported language, built into the C++ class.

This design greatly improved the extensibility of Genparse. In order to support a new output language, one only needs to add appropriate member functions to the C++ class. This implementation became version 0.4.

An important part of 0.4 was support for C++ as an output language. Rather than returning a `struct` containing the command line parameter values, a C++ command line parser would be encapsulated by a C++ class. The user would call the member functions of this class to access parameter values, making for a cleaner interface than C can support.

Version 0.4 also included a more comprehensive set of test suites along with the documentation that you currently are reading.

In Fall 2000, I revisited Genparse. Version 0.4 did not support options with no short form, and thus was limited to 52 options at most. Version 0.5 lifted this restriction, included `<stdlib.h>` rather than the depreciated `<malloc.h>` in output files, and fixed up user-defined include files so that they worked with quotes as well as `"<>"`. Also, I finally figured out how to get Genparse to reliably compile on systems both with and without the `getopt_long ()` function.

Version 0.5.1 fixed a few problems with the lexical analyzer that resulted in default values for floats not being used properly. This version also eliminated the mandatory use of the ‘`-q`’ option.

7 Index