Genparse

Command Line Parser Generator
Edition 0.5.1, for Genparse version 0.5.1
11 October 2000

Mike Borella

Copyright © 1997-2000, Mike Borella

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Author.

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++, 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.
- 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.

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.sourgeforge.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", 0, 0, 'h'},
    {"iterations", 1, 0, 'i'},
    {0, 0, 0, 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 */
#mandatory file
i / iterations  int 1   [0...]  "Number of times to output <file>."
o / outfile     string  {""} "Output file."
```

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 indicates that there is a "mandatory" argument for `mycopy3`, that being the file to copy. The `#mandatory` directive does not actually check whether or not the argument exists - it only is used to create friendlier output for the `usage()` function.

The third 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 fourth 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.

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 23.), 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 q;
  bool v;
  int optind;
};

/* function prototypes */
struct arg_t * Cmdline(int, char **);
void usage(char *);
void free_args(struct arg_t *);
```

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 native C++ string type 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], the usage function, and the `free_args()` function. The `Cmdline()` function is where the meat of Genparse processing occurs. It takes as arguments `argc` and `argv`, and returns the structure containing the values of the options. 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

[2] The name of the command line parsing function can be user-defined. See Chapter 3 [Genparse Options], page 23.

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.

The free_args() function should be called if the main program no longer needs to use the struct arg_t structure. This function will deallocate all memory associated with the structure and its fields.

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 <unistd.h>
#include <stdlib.h>
#include <getopt.h>
#include "mycopy3_clp.h"

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

void usage(char *executable)
{
  printf("usage: %s [ -iohqv ] file \n", executable);
  printf("  [ -i ] ");
  printf("[ --iterations  ] ");
  printf("Number of times to output <file>. ");
  printf(" (default = 1)");
  printf("\n");
  printf("  [ -o ] ");
  printf("[ --outfile  ] ");
  printf("Output file. ");
  printf("\n");
  printf("  [ -h ] ");
  printf("[ --help  ] ");
  printf("Display help information. ");
  printf(" (default = 0)");
  printf("\n");
  printf("  [ -v ] ");
  printf("[ --version  ] ");
  printf("Output version. ");
  printf(" (default = 0)");
  printf("\n");

  exit(1);
}

/*----------------------------------------------------------------------------
**
** free_args()
**
** Call this to free the memory that was dynamically allocated by the parser.
**
**--------------------------------------------------------------------------*/

void free_args(struct arg_t *my_args)
{
  if (my_args->o != NULL) free(my_args->o);
  free(my_args);
}

/*----------------------------------------------------------------------------
**
** Cmdline()
**
** Parse the argv array into the command line structure
**
**--------------------------------------------------------------------------*/

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

  static struct option long_options[] =
  {
    {"iterations", 1, 0, 'i'},
    {"outfile", 1, 0, 'o'},
    {"help", 0, 0, 'h'},
    {"quiet", 0, 0, 'q'},
    {"version", 0, 0, 'v'},
    {0, 0, 0, 0}
  };

  my_args = (struct arg_t *) malloc (sizeof(struct arg_t));

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

  while ((c = getopt_long(argc, argv, "i:o:hqv", long_options,
                          &option_index)) != 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 = strdup(optarg);
          break;

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

        case 'q':
          my_args->q = true;
          break;

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

        default:
          usage(argv[0]);

        }
    } /* while */

  if (errflg)
```

```
      usage(argv[0]);

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

The parser file consists of three main functions. The `usage()` function displays the program's usage information, then terminates. The `free_args()` function frees all memory that was dynamically allocated by the parsing function. The parsing function, named `Cmdline()` in this case[3], reads the command line and returns a `struct arg_t` containing the command line parameters.

Since the first two functions are straightforward, we will not examine them in detail. Instead, we will focus our attention on `Cmdline()`. This function begins by defining a `struct option` 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.
- For '`-o`', a character array is dynamically allocated; thus, the need for the `free_args()` function.

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"
```

[3] The name of the parser function can be set by the user. See Chapter 3 [Genparse Options], page 23.

```
#define VERSION "3.0"

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

  a = Cmdline(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);
  free_args(a);
  return 0;
}
```

The `mycopy3.c` module begins by including `mycopy3_clp.h`, which is necessary for the definition of `struct arg_t` and the parser function prototypes. The `Cmdline()` function is called immediately, and the result is returned to a local `struct arg_t` pointer. 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*.

The program ends by deallocating parameter memory by calling `free_args()`.

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>`. Quotes are not allowed in include directives. 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>
#mandatory file
i / iterations  int 1   [0...MAX]  "Number of times to output <file>."
o / outfile     string  {""} "Output file."
```

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

This file instructs Genparse to create a global callback function `my_callback()` and the option callback function `outfile_cb()`. The `my_callback()` function is called by the parser, while the user determines when to call `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 param_o_callback(char *);
```

In the `mycopy4_clp.h` file, prototypes for the global and the option callback functions. Note that the convention for naming option callback functions is "param_X_callback()" where X is the short form of the option.

```
/* mycopy4_clp.c */
if (!my_callback(my_args))
  usage(argv[0]);
```

In the mycopy4_clp.c file, a call to the global callback is made. If a 0 is returned, an error is assumed and the usage function is called. There may be more than one global callback - all will be called in the parser function.

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;
}

/*----------------------------------------------------------------------------
**
** param_o_callback()
**
** User defined parameter callback.
**
**--------------------------------------------------------------------------*/

int param_o_callback(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;
  string _o;
  bool _h;
  bool _q;
  bool _v;

  /* other stuff to keep track of */
  string _executable;
  int _optind;

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

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

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

  int i() { return _i; }
  string o() { return _o; }
  bool h() { return _h; }
  bool q() { return _q; }
  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 an a private member variable, and must be accessed through a call to a the member function named with the short 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.
- There is no function analogous to `free_args()`. Since the class does not dynamically allocate memory, the destructor will handle all deallocation.
- 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"
#include "mycopy4_clp_cb.cc"

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

```
** Constructor method.
**
**--------------------------------------------------------------------------*/

Cmdline::Cmdline(int argc, char *argv[]) throw (string)
{
  extern char *optarg;
  extern int optind;
  int option_index = 0;
  char c;

  static struct option long_options[] =
  {
    {"iterations", 1, 0, 'i'},
    {"outfile", 1, 0, 'o'},
    {"help", 1, 0, 'h'},
    {"quiet", 1, 0, 'q'},
    {"version", 1, 0, 'v'},
    {0, 0, 0, 0}
  };
  _executable += argv[0];

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

  while ((c = getopt_long(argc, argv, "i:o:hqv", long_options,
                          &option_index)) != EOF)
    {
      switch(c)
        {
        case 'i':
          _i = atoi(optarg);
          if (_i < 1)
            {
              string s;
              s += "parameter range error: i must be >= 1";
              throw(s);
            }
          if (_i > MAX)
            {
              string s;
              s += "parameter range error: i must be <= MAX";
              throw(s);
            }
          break;
        case 'o':
          _o = optarg;
          if (!param_o_callback())
            this->usage();
          break;

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

        case 'q':
          _q = true;
          break;

       case 'v':
          _v = true;
          break;

        default:
          this->usage();

        }
    } /* while */

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

}

/*----------------------------------------------------------------------------
**
** Cmdline::usage()
**
** Usage function.
**
**--------------------------------------------------------------------------*/

void Cmdline::usage()
{
  cout << "usage: " << _executable << " [ -iohqv ]  <file>" << endl;
  cout << "  [ -i ]  ";
  cout << "[ --iterations ]  ";
  cout << "Number of times to output <file>. ";
  cout << "(default = 1)";
  cout << endl;
  cout << "  [ -o ]  ";
  cout << "[ --outfile ]  ";
```

```
    cout << "Output file. ";
    cout << endl;
    cout << "  [ -h ]  ";
    cout << "[ --help ]  ";
    cout << "Display help information. ";
    cout << "(default = 0)";
    cout << endl;
    cout << "  [ -v ]  ";
    cout << "[ --version ]  ";
    cout << "Output version. ";
    cout << "(default = 0)";
    cout << endl;
    exit(0);
  }
```

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()
**
** Global callback.
**
**--------------------------------------------------------------------------*/

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

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

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

```
**
** Cmdline::param_o()
**
** Parameter callback.
**
**--------------------------------------------------------------------------*/

bool Cmdline::param_o_callback()
{
  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 <iostream>
#include <fstream>
#include "mycopy4_clp.h"

#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 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 23.)
- 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 16.

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 `yacc` 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.
- '`-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.
- '`-l`' / '`--language`': Output language. The programming language in which Genparse writes the output files. Currently only C and C++ are supported. The default is C. To indicate C++, the following strings may be used: "c++", "cpp", "cc", and "cxx".
- '`-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".
- '`-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++) that does the actual command line parsing. Default value is "Cmdline".
- '`-v`' / '`--version`': Version. If this option is set, the version number is displayed, then the program is terminated. The default value is off.

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 created them with the default behavior.

4 Genparse File Grammar

A Genparse file consists of a number of include and mandatory declarations and global callbacks followed by definitions of parameters. Each option definition must include a short form. The long form, separated from the short form by a slash, is optional. The type of the parameter must follow directly after the short and long forms. The remaining fields, default value, range, callback and description, are optional, and may appear in any order. Only one of each field may be defined per option.

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 `yacc` grammar. Items in capitals are tokens that are defined in the lex file.

```
all: globals entries
        | entries

globals: globals global
        | global

global: include
        | mandatory
        | global_callback

include: #include<FILENAME>

mandatory: #mandatory<FILENAME>

global_callback: callback

entries: entries entry
        | entry

entry: param
        | param type
        | param type options

options: options option
        | option

option: default
        | range
        | callback
        | description

param: short_param
        | NONE / long_param
        | short_param / long_param

short_param: CHAR
```

```
long_param: VAR

type: INT
        | FLOAT
        | STRING
        | CHAR
        | FLAG

default: ALNUM
        | {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
```

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 `yacc` 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 `yacc` 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