My Project

Generated by Doxygen 1.7.6.1

Wed Jan 15 2014 02:16:39

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 option Namespace Reference

The namespace of The Lean Mean C++ Option Parser.

Classes

- struct Descriptor

  Describes an option, its help text (usage) and how it should be parsed.
- class Option

  A parsed option from the command line together with its argument if it has one.
- struct Arg

  Functions for checking the validity of option arguments.
- struct Stats

  Determines the minimum lengths of the buffer and options arrays used for Parser.
- class Parser

  Checks argument vectors for validity and parses them into data structures that are easier to work with.
- struct PrintUsageImplementation

Typedefs

- typedef ArgStatus(∗ CheckArg)(const Option &option, bool msg)

  Signature of functions that check if an argument is valid for a certain type of option.

Enumerations

- enum ArgStatus { ARG_NONE, ARG_OK, ARG_IGNORE, ARG_ILLEGAL }

  Possible results when checking if an argument is valid for a certain option.

Functions

- template<typename OStream >
  void printUsage (OStream &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)

  Outputs a nicely formatted usage string with support for multi-column formatting and line-wrapping.
- template<typename Function >
  void **printUsage** (Function *prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Temporary >
  void **printUsage** (const Temporary &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Syscall >
  void **printUsage** (Syscall *prn, int fd, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Function , typename Stream >
  void **printUsage** (Function *prn, Stream *stream, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)

5.1.1 Detailed Description

The namespace of The Lean Mean C++ Option Parser.

5.1.2 Typedef Documentation

5.1.2.1 typedef ArgStatus(* option::CheckArg)(const Option &option, bool msg)

Signature of functions that check if an argument is valid for a certain type of option.

Every Option has such a function assigned in its Descriptor.

```
Descriptor usage[] = { {UNKNOWN, 0, "", "", Arg::None, ""}, ... };
```

A CheckArg function has the following signature:

```
ArgStatus CheckArg(const Option& option, bool msg);
```

It is used to check if a potential argument would be acceptable for the option. It will even be called if there is no argument. In that case `option.arg` will be `NULL`.

If `msg` is `true` and the function determines that an argument is not acceptable and that this is a fatal error, it should output a message to the user before returning ARG_ILLEGAL. If `msg` is `false` the function should remain silent (or you will get duplicate messages).

See ArgStatus for the meaning of the return values.

While you can provide your own functions, often the following pre-defined checks (which never return ARG_ILLEGAL) will suffice:

- `Arg::None` For options that don't take an argument: Returns ARG_NONE.
- `Arg::Optional` Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

5.1.3 Enumeration Type Documentation

5.1.3.1 enum option::ArgStatus

Possible results when checking if an argument is valid for a certain option.

In the case that no argument is provided for an option that takes an optional argument, return codes `ARG_OK` and `ARG_IGNORE` are equivalent.

Enumerator:

ARG_NONE The option does not take an argument.

ARG_OK The argument is acceptable for the option.

ARG_IGNORE The argument is not acceptable but that's non-fatal because the option's argument is optional.

ARG_ILLEGAL The argument is not acceptable and that's fatal.

5.1.4 Function Documentation

5.1.4.1 template<typename OStream > void option::printUsage (OStream & *prn*, const Descriptor *usage[]*, int *width* = `80`, int *last_column_min_percent* = `50`, int *last_column_own_line_max_percent* = `75`)

Outputs a nicely formatted usage string with support for multi-column formatting and line-wrapping.

printUsage() takes the `help` texts of a Descriptor[] array and formats them into a usage message, wrapping lines to achieve the desired output width.

Table formatting:

Aside from plain strings which are simply line-wrapped, the usage may contain tables. Tables are used to align elements in the output.

```
// Without a table. The explanatory texts are not aligned.
-c, --create  |Creates something.
-k, --kill  |Destroys something.

// With table formatting. The explanatory texts are aligned.
-c, --create  |Creates something.
-k, --kill    |Destroys something.
```

Table formatting removes the need to pad help texts manually with spaces to achieve alignment. To create a table, simply insert \t (tab) characters to separate the cells within a row.

```
const option::Descriptor usage[] = {
{..., "-c, --create  \tCreates something." },
{..., "-k, --kill  \tDestroys something." }, ...
```

Note that you must include the minimum amount of space desired between cells yourself. Table formatting will insert further spaces as needed to achieve alignment.

You can insert line breaks within cells by using \v (vertical tab).

```
const option::Descriptor usage[] = {
{..., "-c,\v--create  \tCreates\vsomething." },
{..., "-k,\v--kill  \tDestroys\vsomething." }, ...

// results in

-c,       Creates
--create  something.
-k,       Destroys
--kill    something.
```

You can mix lines that do not use \t or \v with those that do. The plain lines will not mess up the table layout. Alignment of the table columns will be maintained even across these interjections.

```
const option::Descriptor usage[] = {
{..., "-c, --create  \tCreates something." },
{..., "---------------------------------" },
{..., "-k, --kill  \tDestroys something." }, ...

// results in

-c, --create  Creates something.
---------------------------------
-k, --kill    Destroys something.
```

You can have multiple tables within the same usage whose columns are aligned independently. Simply insert a dummy Descriptor with `help==0`.

```
const option::Descriptor usage[] = {
{..., "Long options:" },
{..., "--very-long-option  \tDoes something long." },
{..., "--ultra-super-mega-long-option  \tTakes forever to complete." },
{..., 0 }, // ---------- table break -----------
{..., "Short options:" },
{..., "-s  \tShort." },
{..., "-q  \tQuick." }, ...

// results in

Long options:
--very-long-option              Does something long.
--ultra-super-mega-long-option  Takes forever to complete.
Short options:
-s  Short.
-q  Quick.

// Without the table break it would be
```

```
Long options:
--very-long-option              Does something long.
--ultra-super-mega-long-option  Takes forever to complete.
Short options:
-s                              Short.
-q                              Quick.
```

Output methods:

Because TheLeanMeanC++Option parser is freestanding, you have to provide the means for output in the first argument(s) to printUsage(). Because printUsage() is implemented as a set of template functions, you have great flexibility in your choice of output method. The following example demonstrates typical uses. Anything that's similar enough will work.

```
#include <unistd.h>  // write()
#include <iostream>  // cout
#include <sstream>   // ostringstream
#include <cstdio>    // fwrite()
using namespace std;

void my_write(const char* str, int size) {
  fwrite(str, size, 1, stdout);
}

struct MyWriter {
  void write(const char* buf, size_t size) const {
     fwrite(str, size, 1, stdout);
  }
};

struct MyWriteFunctor {
  void operator()(const char* buf, size_t size) {
     fwrite(str, size, 1, stdout);
  }
};
...
printUsage(my_write, usage);    // custom write function
printUsage(MyWriter(), usage);  // temporary of a custom class
MyWriter writer;
printUsage(writer, usage);      // custom class object
MyWriteFunctor wfunctor;
printUsage(&wfunctor, usage);   // custom functor
printUsage(write, 1, usage);    // write() to file descriptor 1
printUsage(cout, usage);        // an ostream&
printUsage(fwrite, stdout, usage);  // fwrite() to stdout
ostringstream sstr;
printUsage(sstr, usage);        // an ostringstream&
```

Notes:

- the `write()` method of a class that is to be passed as a temporary as `MyWriter()` is in the example, must be a `const` method, because temporary objects are passed as const reference. This only applies to temporary objects that are created and destroyed in the same statement. If you create an object like `writer` in the example, this restriction does not apply.
- a functor like `MyWriteFunctor` in the example must be passed as a

pointer. This differs from the way functors are passed to e.g. the STL algorithms.

- All printUsage() templates are tiny wrappers around a shared non-template implementation. So there's no penalty for using different versions in the same program.
- printUsage() always interprets Descriptor::help as UTF-8 and always produces UTF-8-encoded output. If your system uses a different charset, you must do your own conversion. You may also need to change the font of the console to see non-ASCII characters properly. This is particularly true for Windows.
- **Security warning:** Do not insert untrusted strings (such as user-supplied arguments) into the usage. printUsage() has no protection against malicious UTF-8 sequences.

Parameters

prn	The output method to use. See the examples above.
usage	the Descriptor[] array whose `help` texts will be formatted.
width	the maximum number of characters per output line. Note that this number is in actual characters, not bytes. printUsage() supports UTF-8 in `help` and will count multi-byte UTF-8 sequences properly. Asian wide characters are counted as 2 characters.
last_column_min_percent	(0-100) The minimum percentage of `width` that should be available for the last column (which typically contains the textual explanation of an option). If less space is available, the last column will be printed on its own line, indented according to `last_column_own_line_max_percent`.
last_column_own_line_max_percent	(0-100) If the last column is printed on its own line due to less than `last_column_min_percent` of the width being available, then only `last_column_own_line_max_percent` of the extra line(s) will be used for the last column's text. This ensures an indentation. See example below.

```
// width=20, last_column_min_percent=50 (i.e. last col. min. width=10)
--3456789 1234567890
          1234567890

// width=20, last_column_min_percent=75 (i.e. last col. min. width=15)
// last_column_own_line_max_percent=75
--3456789
     123456789012345
     67890

// width=20, last_column_min_percent=75 (i.e. last col. min. width=15)
// last_column_own_line_max_percent=33 (i.e. max. 5)
--3456789
               12345
               67890
               12345
               67890
```

Chapter 6

Class Documentation

6.1 option::Parser::Action Struct Reference

Inherited by option::Parser::StoreOptionAction, and option::Stats::CountOptionsAction.

Public Member Functions

- virtual bool perform (Option &)

  Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.
- virtual bool finished (int numargs, const char **args)

  Called by Parser::workhorse() after finishing the parse.

6.1.1 Member Function Documentation

6.1.1.1 virtual bool option::Parser::Action::finished (int *numargs*, const char ** *args*) [inline, virtual]

Called by Parser::workhorse() after finishing the parse.

Parameters

numargs	the number of non-option arguments remaining
args	pointer to the first remaining non-option argument (if numargs > 0).

Returns

`false` iff a fatal error has occurred.

Reimplemented in option::Parser::StoreOptionAction.

6.1.1.2 virtual bool option::Parser::Action::perform (Option &) [inline, virtual]

Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.

Returns `false` iff a fatal error has occured and the parse should be aborted.

Reimplemented in option::Parser::StoreOptionAction, and option::Stats::CountOptionsAction.

The documentation for this struct was generated from the following file:

- optionparser.h

6.2 option::Arg Struct Reference

Functions for checking the validity of option arguments.

`#include <optionparser.h>`

Static Public Member Functions

- static ArgStatus None (const Option &, bool)

  For options that don't take an argument: Returns ARG_NONE.
- static ArgStatus Optional (const Option &option, bool)

  Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

6.2.1 Detailed Description

Functions for checking the validity of option arguments.

Every Option has such a function assigned in its Descriptor.

```
Descriptor usage[] = { {UNKNOWN, 0, "", "", Arg::None, ""}, ... };
```

A CheckArg function has the following signature:

```
ArgStatus CheckArg(const Option& option, bool msg);
```

It is used to check if a potential argument would be acceptable for the option. It will even be called if there is no argument. In that case `option.arg` will be `NULL`.

If `msg` is `true` and the function determines that an argument is not acceptable and that this is a fatal error, it should output a message to the user before returning ARG_ILLEGAL. If `msg` is `false` the function should remain silent (or you will get duplicate messages).

See ArgStatus for the meaning of the return values.

While you can provide your own functions, often the following pre-defined checks (which never return ARG_ILLEGAL) will suffice:

- `Arg::None` For options that don't take an argument: Returns ARG_NONE.
- `Arg::Optional` Returns ARG_OK if the argument is attached and ARG_IGNORE otherwise.

The following example code can serve as starting place for writing your own more complex CheckArg functions:

```
struct Arg: public option::Arg
{
  static void printError(const char* msg1, const option::Option& opt, const
     char* msg2)
  {
    fprintf(stderr, "ERROR: %s", msg1);
    fwrite(opt.name, opt.namelen, 1, stderr);
    fprintf(stderr, "%s", msg2);
  }

  static option::ArgStatus Unknown(const option::Option& option, bool msg)
  {
    if (msg) printError("Unknown option '", option, "'\n");
    return option::ARG_ILLEGAL;
  }

  static option::ArgStatus Required(const option::Option& option, bool msg)
  {
    if (option.arg != 0)
      return option::ARG_OK;

    if (msg) printError("Option '", option, "' requires an argument\n");
    return option::ARG_ILLEGAL;
  }

  static option::ArgStatus NonEmpty(const option::Option& option, bool msg)
  {
    if (option.arg != 0 && option.arg[0] != 0)
      return option::ARG_OK;

    if (msg) printError("Option '", option, "' requires a non-empty argument\n
     ");
    return option::ARG_ILLEGAL;
  }

  static option::ArgStatus Numeric(const option::Option& option, bool msg)
  {
    char* endptr = 0;
    if (option.arg != 0 && strtol(option.arg, &endptr, 10)){};
    if (endptr != option.arg && *endptr == 0)
      return option::ARG_OK;

    if (msg) printError("Option '", option, "' requires a numeric argument\n")
     ;
    return option::ARG_ILLEGAL;
  }
};
```

The documentation for this struct was generated from the following file:

- optionparser.h

6.3 option::Stats::CountOptionsAction Class Reference

Inherits option::Parser::Action.

Public Member Functions

- CountOptionsAction (unsigned *buffer_max_)
- bool perform (Option &)

  Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.

6.3.1 Constructor & Destructor Documentation

6.3.1.1 option::Stats::CountOptionsAction::CountOptionsAction (unsigned * *buffer_max_*) [inline]

Creates a new CountOptionsAction that will increase `*buffer_max_` for each parsed Option.

6.3.2 Member Function Documentation

6.3.2.1 bool option::Stats::CountOptionsAction::perform (Option &) [inline, virtual]

Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.

Returns `false` iff a fatal error has occured and the parse should be aborted.

Reimplemented from option::Parser::Action.

The documentation for this class was generated from the following file:

- optionparser.h

6.4 option::Descriptor Struct Reference

Describes an option, its help text (usage) and how it should be parsed.

```
#include <optionparser.h>
```

Public Attributes

- const unsigned index

  Index of this option's linked list in the array filled in by the parser.
- const int type

  Used to distinguish between options with the same index. See index for details.
- const char *const shortopt

  Each char in this string will be accepted as a short option character.
- const char *const longopt

  The long option name (without the leading --).
- const CheckArg check_arg

  For each option that matches shortopt or longopt this function will be called to check a potential argument to the option.
- const char * help

  The usage text associated with the options in this Descriptor.

6.4.1 Detailed Description

Describes an option, its help text (usage) and how it should be parsed.

The main input when constructing an option::Parser is an array of Descriptors.

Example:

```
enum OptionIndex {CREATE, ...};
enum OptionType {DISABLE, ENABLE, OTHER};

const option::Descriptor usage[] = {
  { CREATE,                                              // index
    OTHER,                                               // type
    "c",                                                 // shortopt
    "create",                                            // longopt
    Arg::None,                                           // check_arg
    "--create  Tells the program to create something." // help
  }
  , ...
};
```

6.4.2 Member Data Documentation

6.4.2.1 const CheckArg option::Descriptor::check_arg

For each option that matches shortopt or longopt this function will be called to check a potential argument to the option.

This function will be called even if there is no potential argument. In that case it will be passed `NULL` as `arg` parameter. Do not confuse this with the empty string.

See CheckArg for more information.

6.4.2.2 const char* option::Descriptor::help

The usage text associated with the options in this Descriptor.

You can use option::printUsage() to format your usage message based on the `help` texts. You can use dummy Descriptors where shortopt and longopt are both the empty string to add text to the usage that is not related to a specific option.

See option::printUsage() for special formatting characters you can use in `help` to get a column layout.

Attention

Must be UTF-8-encoded. If your compiler supports C++11 you can use the "u8" prefix to make sure string literals are properly encoded.

6.4.2.3 const unsigned option::Descriptor::index

Index of this option's linked list in the array filled in by the parser.

Command line options whose Descriptors have the same index will end up in the same linked list in the order in which they appear on the command line. If you have multiple long option aliases that refer to the same option, give their descriptors the same `index`.

If you have options that mean exactly opposite things (e.g. `--enable-foo` and `--disable-foo`), you should also give them the same `index`, but distinguish them through different values for type. That way they end up in the same list and you can just take the last element of the list and use its type. This way you get the usual behaviour where switches later on the command line override earlier ones without having to code it manually.

Tip:

Use an enum rather than plain ints for better readability, as shown in the example at Descriptor.

6.4.2.4 const char* const option::Descriptor::longopt

The long option name (without the leading `--`).

If this Descriptor should not have a long option name, use the empty string "". NULL is not permitted here!

While shortopt allows multiple short option characters, each Descriptor can have only a single long option name. If you have multiple long option names referring to the same option use separate Descriptors that have the same index and type. You may repeat short option characters in such an alias Descriptor but there's no need to.

Dummy Descriptors:

You can use dummy Descriptors with an empty string for both shortopt and longopt

to add text to the usage that is not related to a specific option. See help. The first dummy Descriptor will be used for unknown options (see below).

Unknown Option Descriptor:

The first dummy Descriptor in the list of Descriptors, whose shortopt and longopt are both the empty string, will be used as the Descriptor for unknown options. An unknown option is a string in the argument vector that is not a lone minus '-' but starts with a minus character and does not match any Descriptor's shortopt or longopt.
Note that the dummy descriptor's check_arg function *will* be called and its return value will be evaluated as usual. I.e. if it returns ARG_ILLEGAL the parsing will be aborted with `Parser::error()==true`.
if `check_arg` does not return ARG_ILLEGAL the descriptor's index *will* be used to pick the linked list into which to put the unknown option.
If there is no dummy descriptor, unknown options will be dropped silently.

6.4.2.5 const char* const option::Descriptor::shortopt

Each char in this string will be accepted as a short option character.

The string must not include the minus character '-' or you'll get undefined behaviour.

If this Descriptor should not have short option characters, use the empty string "". NULL is not permitted here!

See longopt for more information.

6.4.2.6 const int option::Descriptor::type

Used to distinguish between options with the same index. See index for details.

It is recommended that you use an enum rather than a plain int to make your code more readable.

The documentation for this struct was generated from the following file:

- optionparser.h

6.5 option::PrintUsageImplementation::FunctionWriter< Function > Struct Template Reference

Inherits option::PrintUsageImplementation::IStringWriter.

Public Member Functions

- virtual void operator() (const char *str, int size)

Writes the given number of chars beginning at the given pointer somewhere.

- **FunctionWriter** (Function *w)

Public Attributes

- Function * **write**

template<typename Function> struct option::PrintUsageImplementation::FunctionWriter< Function >

The documentation for this struct was generated from the following file:

- optionparser.h

6.6 option::PrintUsageImplementation::IStringWriter Struct Reference

Inherited by option::PrintUsageImplementation::FunctionWriter< Function >, option::PrintUsageImplementation::OStreamWriter< OStream >, option::PrintUsageImplementation::StreamWriter< Function, Stream >, option::PrintUsageImplementation::SyscallWriter< Syscall >, and option::PrintUsageImplementation::TemporaryWriter< Temporary >.

Public Member Functions

- virtual void operator() (const char *, int)

  Writes the given number of chars beginning at the given pointer somewhere.

The documentation for this struct was generated from the following file:

- optionparser.h

6.7 option::PrintUsageImplementation::LinePartIterator Class Reference

Public Member Functions

- LinePartIterator (const Descriptor usage[])

  Creates an iterator for `usage`.
- bool nextTable ()

  Moves iteration to the next table (if any). Has to be called once on a new LinePartIterator to move to the 1st table.

- void restartTable ()

  Reset iteration to the beginning of the current table.
- bool nextRow ()

  Moves iteration to the next row (if any). Has to be called once after each call to nextTable() to move to the 1st row of the table.
- void restartRow ()

  Reset iteration to the beginning of the current row.
- bool next ()

  Moves iteration to the next part (if any). Has to be called once after each call to nextRow() to move to the 1st part of the row.
- int column ()

  Returns the index (counting from 0) of the column in which the part pointed to by data() is located.
- int line ()

  Returns the index (counting from 0) of the line within the current column this part belongs to.
- int length ()

  Returns the length of the part pointed to by data() in raw chars (not UTF-8 characters).
- int screenLength ()

  Returns the width in screen columns of the part pointed to by data(). Takes multi-byte UTF-8 sequences and wide characters into account.
- const char * data ()

  Returns the current part of the iteration.

6.7.1 Member Function Documentation

6.7.1.1 bool option::PrintUsageImplementation::LinePartIterator::next ( ) [inline]

Moves iteration to the next part (if any). Has to be called once after each call to nextRow() to move to the 1st part of the row.

Return values

false	if moving to next part failed because no further part exists.

See LinePartIterator for details about the iteration.

6.7.1.2 bool option::PrintUsageImplementation::LinePartIterator::nextRow ( ) [inline]

Moves iteration to the next row (if any). Has to be called once after each call to nextTable() to move to the 1st row of the table.

Return values

false	if moving to next row failed because no further row exists.

6.7.1.3 bool option::PrintUsageImplementation::LinePartIterator::nextTable ( ) [inline]

Moves iteration to the next table (if any). Has to be called once on a new LinePartIterator to move to the 1st table.

Return values

false	if moving to next table failed because no further table exists.

The documentation for this class was generated from the following file:

- optionparser.h

6.8 option::PrintUsageImplementation::LineWrapper Class - Reference

Public Member Functions

- void flush (IStringWriter &write)

  Writes out all remaining data from the LineWrapper using `write`. Unlike process() this method indents all lines including the first and will output a \n at the end (but only if something has been written).
- void process (IStringWriter &write, const char *data, int len)

  Process, wrap and output the next piece of data.
- LineWrapper (int x1, int x2)

  Constructs a LineWrapper that wraps its output to fit into screen columns `x1` (incl.) to `x2` (excl.).

6.8.1 Constructor & Destructor Documentation

6.8.1.1 option::PrintUsageImplementation::LineWrapper::LineWrapper (int *x1*, int *x2*) [inline]

Constructs a LineWrapper that wraps its output to fit into screen columns `x1` (incl.) to `x2` (excl.).

`x1` gives the indentation LineWrapper uses if it needs to indent.

6.8.2 Member Function Documentation

6.8.2.1 void option::PrintUsageImplementation::LineWrapper::process (IStringWriter & *write*, const char * *data*, int *len*) [inline]

Process, wrap and output the next piece of data.

process() will output at least one line of output. This is not necessarily the `data` passed in. It may be data queued from a prior call to process(). If the internal buffer is full, more than 1 line will be output.

process() assumes that the a proper amount of indentation has already been output. It won't write any further indentation before the 1st line. If more than 1 line is written due to buffer constraints, the lines following the first will be indented by this method, though.

No \n is written by this method after the last line that is written.

Parameters

write	where to write the data.
data	the new chunk of data to write.
len	the length of the chunk of data to write.

The documentation for this class was generated from the following file:

- optionparser.h

6.9 option::Option Class Reference

A parsed option from the command line together with its argument if it has one.

`#include <optionparser.h>`

Public Member Functions

- int type () const

  Returns Descriptor::type of this Option's Descriptor, or 0 if this Option is invalid (unused).
- int index () const

  Returns Descriptor::index of this Option's Descriptor, or -1 if this Option is invalid (unused).
- int count ()

  Returns the number of times this Option (or others with the same Descriptor::index) occurs in the argument vector.
- bool isFirst () const

  Returns true iff this is the first element of the linked list.
- bool isLast () const

  Returns true iff this is the last element of the linked list.
- Option * first ()

  Returns a pointer to the first element of the linked list.

- Option * last ()

  Returns a pointer to the last element of the linked list.
- Option * prev ()

  Returns a pointer to the previous element of the linked list or NULL if called on first().
- Option * prevwrap ()

  Returns a pointer to the previous element of the linked list with wrap-around from first() to last().
- Option * next ()

  Returns a pointer to the next element of the linked list or NULL if called on last().
- Option * nextwrap ()

  Returns a pointer to the next element of the linked list with wrap-around from last() to first().
- void append (Option *new_last)

  Makes `new_last` the new last() by chaining it into the list after last().
- operator const Option * () const

  Casts from Option to const Option but only if this Option is valid.*
- operator Option * ()

  Casts from Option to Option but only if this Option is valid.*
- Option ()

  Creates a new Option that is a one-element linked list and has NULL desc, name, arg and namelen.
- Option (const Descriptor *desc_, const char *name_, const char *arg_)

  Creates a new Option that is a one-element linked list and has the given values for desc, name and arg.
- void operator= (const Option &orig)

  Makes `*this` a copy of `orig` except for the linked list pointers.
- Option (const Option &orig)

  Makes `*this` a copy of `orig` except for the linked list pointers.

Public Attributes

- const Descriptor * desc

  Pointer to this Option's Descriptor.
- const char * name

  The name of the option as used on the command line.
- const char * arg

  Pointer to this Option's argument (if any).
- int namelen

  The length of the option name.

6.9.1 Detailed Description

A parsed option from the command line together with its argument if it has one.

The Parser chains all parsed options with the same Descriptor::index together to form a linked list. This allows you to easily implement all of the common ways of handling repeated options and enable/disable pairs.

- Test for presence of a switch in the argument vector:

```
if ( options[QUIET] ) ...
```

- Evaluate --enable-foo/--disable-foo pair where the last one used wins:

```
if ( options[FOO].last()->type() == DISABLE ) ...
```

- Cumulative option (-v verbose, -vv more verbose, -vvv even more verbose):

```
int verbosity = options[VERBOSE].count();
```

- Iterate over all --file=<fname> arguments:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
  fname = opt->arg; ...
```

6.9.2 Constructor & Destructor Documentation

6.9.2.1 option::Option::Option (const Descriptor * *desc_*, const char * *name_*, const char * *arg_*) [inline]

Creates a new Option that is a one-element linked list and has the given values for desc, name and arg.

If `name_` points at a character other than '-' it will be assumed to refer to a short option and namelen will be set to 1. Otherwise the length will extend to the first '=' character or the string's 0-terminator.

6.9.2.2 option::Option::Option (const Option & *orig*) [inline]

Makes `*this` a copy of `orig` except for the linked list pointers.

After this operation `*this` will be a one-element linked list.

6.9.3 Member Function Documentation

6.9.3.1 void option::Option::append (Option * *new_last*) [inline]

Makes `new_last` the new last() by chaining it into the list after last().

It doesn't matter which element you call append() on. The new element will always be appended to last().

Attention

`new_last` must not yet be part of a list, or that list will become corrupted, because this method does not unchain `new_last` from an existing list.

6.9.3.2 int option::Option::count ( ) [inline]

Returns the number of times this Option (or others with the same Descriptor::index) occurs in the argument vector.

This corresponds to the number of elements in the linked list this Option is part of. It doesn't matter on which element you call count(). The return value is always the same.

Use this to implement cumulative options, such as -v, -vv, -vvv for different verbosity levels.

Returns 0 when called for an unused/invalid option.

6.9.3.3 Option* option::Option::first ( ) [inline]

Returns a pointer to the first element of the linked list.

Use this when you want the first occurrence of an option on the command line to take precedence. Note that this is not the way most programs handle options. You should probably be using last() instead.

Note

This method may be called on an unused/invalid option and will return a pointer to the option itself.

6.9.3.4 bool option::Option::isFirst ( ) const [inline]

Returns true iff this is the first element of the linked list.

The first element in the linked list is the first option on the command line that has the respective Descriptor::index value.

Returns true for an unused/invalid option.

6.9.3.5 bool option::Option::isLast ( ) const [inline]

Returns true iff this is the last element of the linked list.

The last element in the linked list is the last option on the command line that has the respective Descriptor::index value.

Returns true for an unused/invalid option.

6.9.3.6 Option* option::Option::last ( ) [inline]

Returns a pointer to the last element of the linked list.

Use this when you want the last occurrence of an option on the command line to take precedence. This is the most common way of handling conflicting options.

Note

This method may be called on an unused/invalid option and will return a pointer to the option itself.

Tip:

If you have options with opposite meanings (e.g. `--enable-foo` and `--disable-foo`), you can assign them the same Descriptor::index to get them into the same list. Distinguish them by Descriptor::type and all you have to do is check `last()->type()` to get the state listed last on the command line.

6.9.3.7 Option* option::Option::next ( ) [inline]

Returns a pointer to the next element of the linked list or NULL if called on last().

If called on last() this method returns NULL. Otherwise it will return the option with the same Descriptor::index that follows this option on the command line.

6.9.3.8 Option* option::Option::nextwrap ( ) [inline]

Returns a pointer to the next element of the linked list with wrap-around from last() to first().

If called on last() this method returns first(). Otherwise it will return the option with the same Descriptor::index that follows this option on the command line.

6.9.3.9 option::Option::operator const Option * ( ) const [inline]

Casts from Option to const Option* but only if this Option is valid.

If this Option is valid (i.e. `desc!=NULL`), returns this. Otherwise returns NULL. This allows testing an Option directly in an if-clause to see if it is used:

```
if (options[CREATE])
{
  ...
}
```

It also allows you to write loops like this:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
  fname = opt->arg; ...
```

6.9.3.10 option::Option::operator Option * ( ) [inline]

Casts from Option to Option* but only if this Option is valid.

If this Option is valid (i.e. `desc!=NULL`), returns this. Otherwise returns NULL. This allows testing an Option directly in an if-clause to see if it is used:

```
if (options[CREATE])
{
  ...
}
```

It also allows you to write loops like this:

```
for (Option* opt = options[FILE]; opt; opt = opt->next())
  fname = opt->arg; ...
```

6.9.3.11 void option::Option::operator= (const Option & *orig*) [inline]

Makes `*this` a copy of `orig` except for the linked list pointers.

After this operation `*this` will be a one-element linked list.

6.9.3.12 Option* option::Option::prev ( ) [inline]

Returns a pointer to the previous element of the linked list or NULL if called on first().

If called on first() this method returns NULL. Otherwise it will return the option with the same Descriptor::index that precedes this option on the command line.

6.9.3.13 Option* option::Option::prevwrap ( ) [inline]

Returns a pointer to the previous element of the linked list with wrap-around from first() to last().

If called on first() this method returns last(). Otherwise it will return the option with the same Descriptor::index that precedes this option on the command line.

6.9.3.14 int option::Option::type ( ) const [inline]

Returns Descriptor::type of this Option's Descriptor, or 0 if this Option is invalid (unused).

Because this method (and last(), too) can be used even on unused Options with desc==0, you can (provided you arrange your types properly) switch on type() without testing validity first.

```
enum OptionType { UNUSED=0, DISABLED=0, ENABLED=1 };
enum OptionIndex { FOO };
const Descriptor usage[] = {
  { FOO, ENABLED,  "", "enable-foo",  Arg::None, 0 },
```

```
  { FOO, DISABLED, "", "disable-foo", Arg::None, 0 },
  { 0, 0, 0, 0, 0, 0 } };
...
switch(options[FOO].last()->type()) // no validity check required!
{
  case ENABLED: ...
  case DISABLED: ...  // UNUSED==DISABLED !
}
```

6.9.4 Member Data Documentation

6.9.4.1 const char* option::Option::arg

Pointer to this Option's argument (if any).

NULL if this option has no argument. Do not confuse this with the empty string which is a valid argument.

6.9.4.2 const Descriptor* option::Option::desc

Pointer to this Option's Descriptor.

Remember that the first dummy descriptor (see Descriptor::longopt) is used for unknown options.

Attention

`desc==NULL` signals that this Option is unused. This is the default state of elements in the result array. You don't need to test `desc` explicitly. You can simply write something like this:

```
if (options[CREATE])
{
  ...
}
```

This works because of `operator const Option*()` .

6.9.4.3 const char* option::Option::name

The name of the option as used on the command line.

The main purpose of this string is to be presented to the user in messages.

In the case of a long option, this is the actual `argv` pointer, i.e. the first character is a '-'. In the case of a short option this points to the option character within the `argv` string.

Note that in the case of a short option group or an attached option argument, this string will contain additional characters following the actual name. Use namelen to filter out the actual option name only.

6.9.4.4 int option::Option::namelen

The length of the option name.

Because name points into the actual `argv` string, the option name may be followed by more characters (e.g. other short options in the same short option group). This value is the number of bytes (not characters!) that are part of the actual name.

For a short option, this length is always 1. For a long option this length is always at least 2 if single minus long options are permitted and at least 3 if they are disabled.

Note

In the pathological case of a minus within a short option group (e.g. `-xf-z`), this length is incorrect, because this case will be misinterpreted as a long option and the name will therefore extend to the string's 0-terminator or a following '=" character if there is one. This is irrelevant for most uses of name and `namelen`. If you really need to distinguish the case of a long and a short option, compare name to the `argv` pointers. A long option's `name` is always identical to one of them, whereas a short option's is never.

The documentation for this class was generated from the following file:

- optionparser.h

6.10 option::PrintUsageImplementation::OStreamWriter< OStream > Struct Template Reference

Inherits option::PrintUsageImplementation::IStringWriter.

Public Member Functions

- virtual void operator() (const char *str, int size)

  Writes the given number of chars beginning at the given pointer somewhere.
- **OStreamWriter** (OStream &o)

Public Attributes

- OStream & **ostream**

template<typename OStream> struct option::PrintUsageImplementation::OStreamWriter< OStream >

The documentation for this struct was generated from the following file:

- optionparser.h

6.11 option::Parser Class Reference

Checks argument vectors for validity and parses them into data structures that are easier to work with.

```
#include <optionparser.h>
```

Classes

- struct Action
- class StoreOptionAction

Public Member Functions

- Parser ()

  Creates a new Parser.
- Parser (bool gnu, const Descriptor usage[], int argc, const char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  Creates a new Parser and immediately parses the given argument vector.
- Parser (bool gnu, const Descriptor usage[], int argc, char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  Parser(...) with non-const argv.
- Parser (const Descriptor usage[], int argc, const char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  POSIX Parser(...) (gnu==false).
- Parser (const Descriptor usage[], int argc, char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  POSIX Parser(...) (gnu==false) with non-const argv.
- void parse (bool gnu, const Descriptor usage[], int argc, const char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  Parses the given argument vector.
- void parse (bool gnu, const Descriptor usage[], int argc, char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  parse() with non-const argv.
- void parse (const Descriptor usage[], int argc, const char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

  POSIX parse() (gnu==false).
- void parse (const Descriptor usage[], int argc, char **argv, Option options[], Option buffer[], int min_abbr_len=0, bool single_minus_longopt=false, int bufmax=-1)

POSIX parse() (gnu==false) with non-const argv.

- int optionsCount ()

  Returns the number of valid Option objects in `buffer[]`.

- int nonOptionsCount ()

  Returns the number of non-option arguments that remained at the end of the most recent parse() that actually encountered non-option arguments.

- const char ** nonOptions ()

  Returns a pointer to an array of non-option arguments (only valid if `nonOptionsCount() > 0`).

- const char * nonOption (int i)

  *Returns **`nonOptions()[i]`** (without checking if i is in range!).*

- bool error ()

  Returns `true` if an unrecoverable error occurred while parsing options.

Friends

- struct **Stats**

6.11.1 Detailed Description

Checks argument vectors for validity and parses them into data structures that are easier to work with.

Example:

```
int main(int argc, char* argv[])
{
  argc-=(argc>0); argv+=(argc>0); // skip program name argv[0] if present
  option::Stats  stats(usage, argc, argv);
  option::Option options[stats.options_max], buffer[stats.buffer_max];
  option::Parser parse(usage, argc, argv, options, buffer);

  if (parse.error())
    return 1;

  if (options[HELP])
  ...
```

6.11.2 Constructor & Destructor Documentation

6.11.2.1 option::Parser::Parser (bool *gnu*, const Descriptor *usage[]*, int *argc*, const char ** *argv*, Option *options[]*, Option *buffer[]*, int *min_abbr_len* = 0, bool *single_minus_longopt* = false, int *bufmax* = -1) [inline]

Creates a new Parser and immediately parses the given argument vector.

Parameters

gnu	if true, parse() will not stop at the first non-option argument. Instead it will reorder arguments so that all non-options are at the end. This is the default behaviour of GNU getopt() but is not conforming to POSIX. Note, that once the argument vector has been reordered, the `gnu` flag will have no further effect on this argument vector. So it is enough to pass `gnu==true` when creating Stats.
usage	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
argc	The number of elements from `argv` that are to be parsed. If you pass -1, the number will be determined automatically. In that case the `argv` list must end with a NULL pointer.
argv	The arguments to be parsed. If you pass -1 as `argc` the last pointer in the `argv` list must be NULL to mark the end.
options	Each entry is the first element of a linked list of Options. Each new option that is parsed will be appended to the list specified by that Option's Descriptor::index. If an entry is not yet used (i.e. the Option is invalid), it will be replaced rather than appended to. The minimum length of this array is the greatest Descriptor::index value that occurs in `usage` *PLUS* ONE.
buffer	Each argument that is successfully parsed (including unknown arguments, if they have a Descriptor whose CheckArg does not return ARG_ILLEGAL) will be stored in this array. parse() scans the array for the first invalid entry and begins writing at that index. You can pass `bufmax` to limit the number of options stored.
min_abbr_len	Passing a value `min_abbr_len > 0` enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. `--foob=10` will be interpreted as if it was `--foobar=10`), as long as the prefix has at least `min_abbr_len` characters (not counting the `--`) and is unambiguous. Be careful if combining `min_abbr_len=1` with `single_minus_longopt=true` because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
single_minus_longopt	Passing `true` for this option allows long options to begin with a single minus. The double minus form will still be recognized. Note that single minus long options take precedence over short options and short option groups. E.g. `-file` would be interpreted as `--file` and not as `-f -i -l -e` (assuming a long option named "`file`" exists).
bufmax	The greatest index in the `buffer`[] array that parse() will write to is `bufmax-1`. If there are more options, they will be processed (in particular their CheckArg will be called) but not stored. If you used Stats::buffer_max to dimension this array, you can pass -1 (or not pass `bufmax` at all) which tells parse() that the buffer is "large enough".

Attention

Remember that `options` and `buffer` store Option *objects*, not pointers. - Therefore it is not possible for the same object to be in both arrays. For those options that are found in both `buffer`[] and `options`[] the respective objects are independent copies. And only the objects in `options`[] are properly linked via Option::next() and Option::prev(). You can iterate over `buffer`[] to process all options in the order they appear in the argument vector, but if you want access to the other Options with the same Descriptor::index, then you *must* access the linked list via `options`[]. You can get the linked list in options from a buffer object via something like `options`[buffer[i].index()].

6.11.3 Member Function Documentation

6.11.3.1 bool option::Parser::error ( ) [inline]

Returns `true` if an unrecoverable error occurred while parsing options.

An illegal argument to an option (i.e. CheckArg returns ARG_ILLEGAL) is an unrecoverable error that aborts the parse. Unknown options are only an error if their CheckArg function returns ARG_ILLEGAL. Otherwise they are collected. In that case if you want to exit the program if either an illegal argument or an unknown option has been passed, use code like this

```
if (parser.error() || options[UNKNOWN])
  exit(1);
```

6.11.3.2 const char** option::Parser::nonOptions ( ) [inline]

Returns a pointer to an array of non-option arguments (only valid if `nonOptionsCount() >0`).

Note

- parse() does not copy arguments, so this pointer points into the actual argument vector as passed to parse().
- As explained at nonOptionsCount() this pointer is only changed by parse() calls that actually encounter non-option arguments. A parse() call that encounters only options, will not change nonOptions().

6.11.3.3 int option::Parser::nonOptionsCount ( ) [inline]

Returns the number of non-option arguments that remained at the end of the most recent parse() that actually encountered non-option arguments.

Note

A parse() that does not encounter non-option arguments will leave this value as well as nonOptions() undisturbed. This means you can feed the Parser a default argument vector that contains non-option arguments (e.g. a default filename). Then you feed it the actual arguments from the user. If the user has supplied at least one non-option argument, all of the non-option arguments from the default disappear and are replaced by the user's non-option arguments. However, if the user does not supply any non-option arguments the defaults will still be in effect.

6.11.3.4 int option::Parser::optionsCount ( ) [inline]

Returns the number of valid Option objects in `buffer[]`.

Note

- The returned value always reflects the number of Options in the buffer[] array used for the most recent call to parse().
- The count (and the buffer[]) includes unknown options if they are collected (see Descriptor::longopt).

6.11.3.5 void option::Parser::parse (bool *gnu*, const Descriptor *usage[]*, int *argc*, const char ** *argv*, Option *options[]*, Option *buffer[]*, int *min_abbr_len* = `0`, bool *single_minus_longopt* = `false`, int *bufmax* = `-1`) [inline]

Parses the given argument vector.

Parameters

gnu	if true, parse() will not stop at the first non-option argument. Instead it will reorder arguments so that all non-options are at the end. This is the default behaviour of GNU getopt() but is not conforming to POSIX. Note, that once the argument vector has been reordered, the `gnu` flag will have no further effect on this argument vector. So it is enough to pass `gnu==true` when creating Stats.
usage	Array of Descriptor objects that describe the options to support. The last entry of this array must have 0 in all fields.
argc	The number of elements from `argv` that are to be parsed. If you pass -1, the number will be determined automatically. In that case the `argv` list must end with a NULL pointer.
argv	The arguments to be parsed. If you pass -1 as `argc` the last pointer in the `argv` list must be NULL to mark the end.
options	Each entry is the first element of a linked list of Options. Each new option that is parsed will be appended to the list specified by that Option's Descriptor::index. If an entry is not yet used (i.e. the Option is invalid), it will be replaced rather than appended to. The minimum length of this array is the greatest Descriptor::index value that occurs in `usage` *PLUS* ONE.

buffer	Each argument that is successfully parsed (including unknown arguments, if they have a Descriptor whose CheckArg does not return ARG_ILLEGAL) will be stored in this array. parse() scans the array for the first invalid entry and begins writing at that index. You can pass `bufmax` to limit the number of options stored.
min_abbr_len	Passing a value `min_abbr_len > 0` enables abbreviated long options. The parser will match a prefix of a long option as if it was the full long option (e.g. `--foob=10` will be interpreted as if it was `--foobar=10`), as long as the prefix has at least `min_abbr_len` characters (not counting the `--`) and is unambiguous. Be careful if combining `min_abbr_len=1` with `single_minus_longopt=true` because the ambiguity check does not consider short options and abbreviated single minus long options will take precedence over short options.
single_minus_longopt	Passing `true` for this option allows long options to begin with a single minus. The double minus form will still be recognized. Note that single minus long options take precedence over short options and short option groups. E.g. `-file` would be interpreted as `--file` and not as `-f -i -l -e` (assuming a long option named `"file"` exists).
bufmax	The greatest index in the `buffer`[] array that parse() will write to is `bufmax-1`. If there are more options, they will be processed (in particular their CheckArg will be called) but not stored. If you used Stats::buffer_max to dimension this array, you can pass -1 (or not pass `bufmax` at all) which tells parse() that the buffer is "large enough".

Attention

Remember that `options` and `buffer` store Option *objects*, not pointers. - Therefore it is not possible for the same object to be in both arrays. For those options that are found in both `buffer`[] and `options`[] the respective objects are independent copies. And only the objects in `options`[] are properly linked via Option::next() and Option::prev(). You can iterate over `buffer`[] to process all options in the order they appear in the argument vector, but if you want access to the other Options with the same Descriptor::index, then you *must* access the linked list via `options`[]. You can get the linked list in options from a buffer object via something like `options`[buffer[i].index()].

The documentation for this class was generated from the following file:

- optionparser.h

6.12 option::PrintUsageImplementation Struct Reference

Classes

- struct FunctionWriter

- struct IStringWriter
- class LinePartIterator
- class LineWrapper
- struct OStreamWriter
- struct StreamWriter
- struct SyscallWriter
- struct TemporaryWriter

Static Public Member Functions

- static void **upmax** (int &i1, int i2)
- static void **indent** (IStringWriter &write, int &x, int want_x)
- static bool isWideChar (unsigned ch)

  Returns true if ch is the unicode code point of a wide character.
- static void **printUsage** (IStringWriter &write, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_-percent=75)

6.12.1 Member Function Documentation

6.12.1.1 static bool option::PrintUsageImplementation::isWideChar (unsigned *ch*) [inline, static]

Returns true if ch is the unicode code point of a wide character.

Note

The following character ranges are treated as wide

```
1100..115F
2329..232A  (just 2 characters!)
2E80..A4C6  except for 303F
A960..A97C
AC00..D7FB
F900..FAFF
FE10..FE6B
FF01..FF60
FFE0..FFE6
1B000......
```

The documentation for this struct was generated from the following file:

- optionparser.h

6.13 option::Stats Struct Reference

Determines the minimum lengths of the buffer and options arrays used for Parser.

```
#include <optionparser.h>
```

Classes

- class CountOptionsAction

Public Member Functions

- Stats ()

  Creates a Stats object with counts set to 1 (for the sentinel element).
- Stats (bool gnu, const Descriptor usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  Creates a new Stats object and immediately updates it for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update options_max.
- Stats (bool gnu, const Descriptor usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  Stats(...) with non-const argv.
- Stats (const Descriptor usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  POSIX Stats(...) (gnu==false).
- Stats (const Descriptor usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  POSIX Stats(...) (gnu==false) with non-const argv.
- void add (bool gnu, const Descriptor usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  Updates this Stats object for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update options_max.
- void add (bool gnu, const Descriptor usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  add() with non-const argv.
- void add (const Descriptor usage[], int argc, const char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  POSIX add() (gnu==false).
- void add (const Descriptor usage[], int argc, char **argv, int min_abbr_len=0, bool single_minus_longopt=false)

  POSIX add() (gnu==false) with non-const argv.

Public Attributes

- unsigned buffer_max

  Number of elements needed for a `buffer[]` array to be used for parsing the same argument vectors that were fed into this Stats object.
- unsigned options_max

  Number of elements needed for an `options[]` array to be used for parsing the same argument vectors that were fed into this Stats object.

6.13.1 Detailed Description

Determines the minimum lengths of the buffer and options arrays used for Parser.

Because Parser doesn't use dynamic memory its output arrays have to be pre-allocated. If you don't want to use fixed size arrays (which may turn out too small, causing command line arguments to be dropped), you can use Stats to determine the correct sizes. Stats work cumulative. You can first pass in your default options and then the real options and afterwards the counts will reflect the union.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 option::Stats::Stats (bool *gnu,* const Descriptor *usage[],* int *argc,* const char ** *argv,* int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

Creates a new Stats object and immediately updates it for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update options_max.

Note

The calls to Stats methods must match the later calls to Parser methods. See Parser::parse() for the meaning of the arguments.

6.13.3 Member Function Documentation

6.13.3.1 void option::Stats::add (bool *gnu,* const Descriptor *usage[],* int *argc,* const char ** *argv,* int *min_abbr_len* = 0, bool *single_minus_longopt* = false) [inline]

Updates this Stats object for the given `usage` and argument vector. You may pass 0 for `argc` and/or `argv`, if you just want to update options_max.

Note

The calls to Stats methods must match the later calls to Parser methods. See Parser::parse() for the meaning of the arguments.

6.13.4 Member Data Documentation

6.13.4.1 unsigned option::Stats::buffer_max

Number of elements needed for a `buffer`[] array to be used for parsing the same argument vectors that were fed into this Stats object.

Note

This number is always 1 greater than the actual number needed, to give you a sentinel element.

6.13.4.2 unsigned option::Stats::options_max

Number of elements needed for an `options[]` array to be used for parsing the same argument vectors that were fed into this Stats object.

Note

- This number is always 1 greater than the actual number needed, to give you a sentinel element.
- This number depends only on the `usage`, not the argument vectors, because the `options` array needs exactly one slot for each possible Descriptor::index.

The documentation for this struct was generated from the following file:

- optionparser.h

6.14 option::Parser::StoreOptionAction Class Reference

Inherits option::Parser::Action.

Public Member Functions

- StoreOptionAction (Parser &parser_, Option options_[], Option buffer_[], int bufmax_)

  Number of slots in `buffer`. `-1` means "large enough".
- bool perform (Option &option)

  Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.
- bool finished (int numargs, const char **args)

  Called by Parser::workhorse() after finishing the parse.

6.14.1 Constructor & Destructor Documentation

6.14.1.1 option::Parser::StoreOptionAction::StoreOptionAction (Parser & *parser_*, Option *options_[]*, Option *buffer_[]*, int *bufmax_*) [inline]

Number of slots in `buffer`. `-1` means "large enough".

Creates a new StoreOption action.

Parameters

*parser_*	the parser whose op_count should be updated.
*options_*	each Option `o` is chained into the linked list `options_`[o.desc->index]
*buffer_*	each Option is appended to this array as long as there's a free slot.
*bufmax_*	number of slots in `buffer_`. `-1` means "large enough".

6.14.2 Member Function Documentation

6.14.2.1 bool option::Parser::StoreOptionAction::finished (int *numargs*, const char ** *args*) [inline, virtual]

Called by Parser::workhorse() after finishing the parse.

Parameters

numargs	the number of non-option arguments remaining
args	pointer to the first remaining non-option argument (if numargs > 0).

Returns

`false` iff a fatal error has occurred.

Reimplemented from option::Parser::Action.

6.14.2.2 bool option::Parser::StoreOptionAction::perform (Option &) [inline, virtual]

Called by Parser::workhorse() for each Option that has been successfully parsed (including unknown options if they have a Descriptor whose Descriptor::check_arg does not return ARG_ILLEGAL.

Returns `false` iff a fatal error has occured and the parse should be aborted.

Reimplemented from option::Parser::Action.

The documentation for this class was generated from the following file:

- optionparser.h

6.15 option::PrintUsageImplementation::StreamWriter< Function, Stream > Struct Template Reference

Inherits option::PrintUsageImplementation::IStringWriter.

Public Member Functions

- virtual void operator() (const char *str, int size)

  Writes the given number of chars beginning at the given pointer somewhere.
- **StreamWriter** (Function *w, Stream *s)

Public Attributes

- Function * **fwrite**
- Stream * **stream**

template<typename Function, typename Stream> struct option::PrintUsageImplementation::StreamWriter< Function, Stream >

The documentation for this struct was generated from the following file:

- optionparser.h

6.16 option::PrintUsageImplementation::SyscallWriter< Syscall > Struct Template Reference

Inherits option::PrintUsageImplementation::IStringWriter.

Public Member Functions

- virtual void operator() (const char *str, int size)

  Writes the given number of chars beginning at the given pointer somewhere.
- **SyscallWriter** (Syscall *w, int f)

Public Attributes

- Syscall * **write**
- int **fd**

template<typename Syscall> struct option::PrintUsageImplementation::SyscallWriter< Syscall >

The documentation for this struct was generated from the following file:

- optionparser.h

6.17 option::PrintUsageImplementation::TemporaryWriter< Temporary > Struct Template Reference

Inherits option::PrintUsageImplementation::IStringWriter.

Public Member Functions

- virtual void operator() (const char *str, int size)

  Writes the given number of chars beginning at the given pointer somewhere.
- **TemporaryWriter** (const Temporary &u)

Public Attributes

- const Temporary & **userstream**

template<typename Temporary> struct option::PrintUsageImplementation::TemporaryWriter< Temporary >

The documentation for this struct was generated from the following file:

- optionparser.h

Chapter 7

File Documentation

7.1 optionparser.h File Reference

This is the only file required to use The Lean Mean C++ Option Parser. Just #include it and you're set.

Classes

- struct option::Descriptor

  Describes an option, its help text (usage) and how it should be parsed.
- class option::Option

  A parsed option from the command line together with its argument if it has one.
- struct option::Arg

  Functions for checking the validity of option arguments.
- struct option::Stats

  Determines the minimum lengths of the buffer and options arrays used for Parser.
- class option::Parser

  Checks argument vectors for validity and parses them into data structures that are easier to work with.
- struct option::Parser::Action
- class option::Stats::CountOptionsAction
- class option::Parser::StoreOptionAction
- struct option::PrintUsageImplementation
- struct option::PrintUsageImplementation::IStringWriter
- struct option::PrintUsageImplementation::FunctionWriter< Function >
- struct option::PrintUsageImplementation::OStreamWriter< OStream >
- struct option::PrintUsageImplementation::TemporaryWriter< Temporary >
- struct option::PrintUsageImplementation::SyscallWriter< Syscall >
- struct option::PrintUsageImplementation::StreamWriter< Function, Stream >
- class option::PrintUsageImplementation::LinePartIterator
- class option::PrintUsageImplementation::LineWrapper

Namespaces

- namespace option

  The namespace of The Lean Mean C++ Option Parser.

Typedefs

- typedef ArgStatus(∗ option::CheckArg)(const Option &option, bool msg)

  Signature of functions that check if an argument is valid for a certain type of option.

Enumerations

- enum option::ArgStatus { option::ARG_NONE, option::ARG_OK, option::ARG_IGNORE, option::ARG_ILLEGAL }

  Possible results when checking if an argument is valid for a certain option.

Functions

- template<typename OStream >
  void option::printUsage (OStream &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)

  Outputs a nicely formatted usage string with support for multi-column formatting and line-wrapping.
- template<typename Function >
  void **option::printUsage** (Function ∗prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Temporary >
  void **option::printUsage** (const Temporary &prn, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Syscall >
  void **option::printUsage** (Syscall ∗prn, int fd, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)
- template<typename Function , typename Stream >
  void **option::printUsage** (Function ∗prn, Stream ∗stream, const Descriptor usage[], int width=80, int last_column_min_percent=50, int last_column_own_line_max_percent=75)

7.1.1 Detailed Description

This is the only file required to use The Lean Mean C++ Option Parser. Just #include it and you're set. The Lean Mean C++ Option Parser handles the program's command line arguments (argc, argv). It supports the short and long option formats of getopt(), getopt_long() and getopt_long_only() but has a more convenient interface. The following features set it apart from other option parsers:

Highlights:

- It is a header-only library. Just `#include "optionparser.h"` and you're set.
- It is freestanding. There are no dependencies whatsoever, not even the C or C++ standard library.
- It has a usage message formatter that supports column alignment and line wrapping. This aids localization because it adapts to translated strings that are shorter or longer (even if they contain Asian wide characters).
- Unlike getopt() and derivatives it doesn't force you to loop through options sequentially. Instead you can access options directly like this:
  - Test for presence of a switch in the argument vector:
    ```
    if ( options[QUIET] ) ...
    ```
  - Evaluate --enable-foo/--disable-foo pair where the last one used wins:
    ```
    if ( options[FOO].last()->type() == DISABLE ) ...
    ```
  - Cumulative option (-v verbose, -vv more verbose, -vvv even more verbose):
    ```
    int verbosity = options[VERBOSE].count();
    ```
  - Iterate over all --file=<fname> arguments:
    ```
    for (Option* opt = options[FILE]; opt; opt = opt->next())
      fname = opt->arg; ...
    ```
  - If you really want to, you can still process all arguments in order:
    ```
    for (int i = 0; i < p.optionsCount(); ++i) {
      Option& opt = buffer[i];
      switch(opt.index()) {
        case HELP:    ...
        case VERBOSE: ...
        case FILE:    fname = opt.arg; ...
        case UNKNOWN: ...
    ```

Despite these features the code size remains tiny. It is smaller than uClibc's GNU getopt() and just a couple 100 bytes larger than uClibc's SUSv3 getopt().
(This does not include the usage formatter, of course. But you don't have to use that.)

Download:

Tarball with examples and test programs: optionparser-1.3.tar.gz
Just the header (this is all you really need): optionparser.h

Changelog:

Version 1.3: Compatible with Microsoft Visual C++.
Version 1.2: Added Option::namelen and removed the extraction of short option characters into a special buffer.
Changed Arg::Optional to accept arguments if they are attached rather than separate. This is what GNU getopt() does and how POSIX recommends utilities should interpret their arguments.
Version 1.1: Optional mode with argument reordering as done by GNU getopt(), so that options and non-options can be mixed. See Parser::parse().

Feedback:

Send questions, bug reports, feature requests etc. to: **optionparser-feedback (a) lists.sourceforge.net**

Example program:

(Note: `option::*` identifiers are links that take you to their documentation.)

```
#include <iostream>
#include "optionparser.h"

enum  optionIndex { UNKNOWN, HELP, PLUS };
const option::Descriptor usage[] =
{
 {UNKNOWN, 0,"" , ""    ,option::Arg::None, "USAGE: example [options]\n\n"
                                            "Options:" },
 {HELP,    0,"" , "help",option::Arg::None, "  --help  \tPrint usage and exit.
     " },
 {PLUS,    0,"p", "plus",option::Arg::None, "  --plus, -p  \tIncrement count."
      },
 {UNKNOWN, 0,"" ,  ""   ,option::Arg::None, "\nExamples:\n"
                                            "  example --unknown --
      --this_is_no_option\n"
                                            "  example -unk --plus -ppp file1
      file2\n" },
 {0,0,0,0,0,0}
};

int main(int argc, char* argv[])
{
  argc-=(argc>0); argv+=(argc>0); // skip program name argv[0] if present
  option::Stats  stats(usage, argc, argv);
  option::Option options[stats.options_max], buffer[stats.buffer_max];
  option::Parser parse(usage, argc, argv, options, buffer);

  if (parse.error())
    return 1;

  if (options[HELP] || argc == 0) {
    option::printUsage(std::cout, usage);
    return 0;
  }

  std::cout << "--plus count: " <<
    options[PLUS].count() << "\n";

  for (option::Option* opt = options[UNKNOWN]; opt; opt = opt->next())
    std::cout << "Unknown option: " << opt->name << "\n";

  for (int i = 0; i < parse.nonOptionsCount(); ++i)
    std::cout << "Non-option #" << i << ": " << parse.nonOption(i) << "\n";
}
```

Option syntax:

- The Lean Mean C++ Option Parser follows POSIX `getopt()` conventions and supports GNU-style `getopt_long()` long options as well as Perl-style single-minus long options (`getopt_long_only()`).
- short options have the format `-X` where `X` is any character that fits in a char.
- short options can be grouped, i.e. `-X -Y` is equivalent to `-XY`.
- a short option may take an argument either separate (`-X foo`) or attached (`-Xfoo`). You can make the parser accept the additional format `-X=foo` by registering `X` as a long option (in addition to being a short option) and enabling single-minus long options.

- an argument-taking short option may be grouped if it is the last in the group, e.g. `-ABCXfoo` or `-ABCX foo` (`foo` is the argument to the `-X` option).
- a lone minus character `'-'` is not treated as an option. It is customarily used where a file name is expected to refer to stdin or stdout.
- long options have the format `--option-name`.
- the option-name of a long option can be anything and include any characters. Even = characters will work, but don't do that.
- [optional] long options may be abbreviated as long as the abbreviation is unambiguous. You can set a minimum length for abbreviations.
- [optional] long options may begin with a single minus. The double minus form is always accepted, too.
- a long option may take an argument either separate (`--option arg`) or attached (`--option=arg`). In the attached form the equals sign is mandatory.
- an empty string can be passed as an attached long option argument: `--option-name=` . Note the distinction between an empty string as argument and no argument at all.
- an empty string is permitted as separate argument to both long and short options.
- Arguments to both short and long options may start with a `'-'` character. E.g. `-X-X` , `-X -X` or `--long-X=-X` . If `-X` and `--long-X` take an argument, that argument will be `"-X"` in all 3 cases.
- If using the built-in Arg::Optional, optional arguments must be attached.
- the special option `--` (i.e. without a name) terminates the list of options. - Everything that follows is a non-option argument, even if it starts with a `'-'` character. The `--` itself will not appear in the parse results.
- the first argument that doesn't start with `'-'` or `'--'` and does not belong to a preceding argument-taking option, will terminate the option list and is the first non-option argument. All following command line arguments are treated as non-option arguments, even if they start with `'-'` .

  NOTE: This behaviour is mandated by POSIX, but GNU getopt() only honours this if it is explicitly requested (e.g. by setting POSIXLY_CORRECT).

  You can enable the GNU behaviour by passing `true` as first argument to e.g. Parser::parse().
- Arguments that look like options (i.e. `'-'` followed by at least 1 character) but aren't, are NOT treated as non-option arguments. They are treated as unknown options and are collected into a list of unknown options for error reporting.
  This means that in order to pass a first non-option argument beginning with the minus character it is required to use the `--` special option, e.g.

  ```
  program -x -- --strange-filename
  ```

  In this example, `--strange-filename` is a non-option argument. If the `--` were omitted, it would be treated as an unknown option.

  See option::Descriptor::longopt for information on how to collect unknown options.