Boost.Program_options

Vladimir Prus

Table of Contents

Introduction

The program_options library allows program developers to obtain *program options*, that is (name, value) pairs from the user, via conventional methods such as command line and config file.

Why would you use such a library, and why is it better than parsing your command line by straightfor-

ward hand-written code?

- It's easier. The syntax for declaring options is simple, and the library itself is small. Things like conversion of option values to desired type and storing into program variables are handled automatically.
- Error reporting is better. All the problems with the command line are reported, while hand-written code can just misparse the input. In addition, the usage message can be automatically generated, to avoid falling out of sync with the real list of options.
- Options can be read from anywhere. Sooner or later the command line will be not enough for your users, and you'll want config files or maybe even environment variables. These can be added without significant effort on your part.

Now let's see some examples of the library usage in the the section called “Tutorial”.

Tutorial

In this section, we'll take a look at the most common usage scenarios of the program_options library, starting with the simplest one. The examples show only the interesting code parts, but the complete programs can be found in the "BOOST_ROOT/libs/program_options/example" directory. Through all the examples, we'll assume that the following namespace alias is in effect:

```
namespace po = boost::program_options;
```

Getting Started

The first example is the simplest possible: it only handles two options. Here's the source code (the full program is in "example/first.cpp"):

```
// Declare the supported options.
po::options_description desc("Allowed options");
desc.add_options()
    ("help", "produce help message")
    ("compression", po::value<int>(), "set compression level")
;

po::variables_map vm;
po::store(po::parse_command_line(ac, av, desc), vm);
po::notify(vm);

if (vm.count("help")) {
    cout << desc << "\n";
    return 1;
}

if (vm.count("compression")) {
    cout << "Compression level was set to "
 << vm["compression"].as<int>() << ".\n";
} else {
    cout << "Compression level was not set.\n";
}
```

We start by declaring all allowed options using the `options_description` class. The

`add_options` method of that class returns a special proxy object that defines `operator()`. Calls to that operator actually declare options. The parameters are option name, information about value, and option description. In this example, the first option has no value, and the second one has a value of type `int`.

After that, an object of class `variables_map` is declared. That class is intended to store values of options, and can store values of arbitrary types. Next, the calls to `store`, `parse_command_line` and `notify` functions cause `vm` to contain all the options found on the command line.

And now, finally, we can use the options as we like. The `variables_map` class can be used just like `std::map`, except that values stored there must be retrieved with the `as` method shown above. (If the type specified in the call to the `as` method is different from the actually stored type, an exception is thrown.)

It's now a good time to try compiling the code yourself, but if you're not yet ready, here's an example session:

```
$bin/gcc/debug/first
Compression level was not set.
$bin/gcc/debug/first --help
Allowed options:
  --help                : produce help message
  --compression arg     : set compression level
$bin/gcc/debug/first --compression 10
Compression level was set to 10.
```

Option Details

An option value, surely, can have other types than `int`, and can have other interesting properties, which we'll discuss right now. The complete version of the code snipped below can be found in "example/options_description.cpp".

Imagine we're writing a compiler. It should take the optimization level, a number of include paths, and a number of input files, and perform some interesting work. Let's describe the options:

```
int opt;
po::options_description desc("Allowed options");
desc.add_options()
    ("help", "produce help message")
    ("optimization", po::value<int>(&opt)->default_value(10),
  "optimization level")
    ("include-path,I", po::value< vector<string> >(),
  "include path")
    ("input-file", po::value< vector<string> >(), "input file")
;
```

The "--help" option should be familiar from the previous example. It's a good idea to have this option in all cases.

The "optimization" option shows two new features. First, we specify the address of the variable(`&opt`). After storing values, that variable will have the value of the option. Second, we specify a default value of 10, which will be used if no value is specified by the user.

The "include-path" option is an example of the only case where the interface of the `options_description` class serves only one source -- the command line. Users typically like to use short option names for common options, and the "include-path,I" name specifies that short option name

is "I". So, both "--include-path" and "-I" can be used.

The "input-file" option specifies the list of files to process. That's okay for a start, but, of course, writing something like:

```
compiler --input-file=a.cpp
```

is a little non-standard, compared with

```
compiler a.cpp
```

We'll address this in a moment.

The command line tokens which have no option name, as above, are called "positional options" by this library. They can be handled too. With a little help from the user, the library can decide that "a.cpp" really means the same as "--input-file=a.cpp". Here's the additional code we need:

```
po::positional_options_description p;
p.add("input-file", -1);

po::variables_map vm;
po::store(po::command_line_parser(ac, av).
          options(desc).positional(p).run(), vm);
po::notify(vm);
```

The first two lines say that all positional options should be translated into "input-file" options. Also note that we use the `command_line_parser` class to parse the command line, not the `parse_command_line` function. The latter is a convenient wrapper for simple cases, but now we need to pass additional information.

By now, all options are described and parsed. We'll save ourselves the trouble of implementing the rest of the compiler logic and only print the options:

```
if (vm.count("include-path"))
{
    cout << "Include paths are: "
         << vm["include-path"].as< vector<string> >() << "\n";
}

if (vm.count("input-file"))
{
    cout << "Input files are: "
         << vm["input-file"].as< vector<string> >() << "\n";
}

cout << "Optimization level is " << opt << "\n";
```

Here's an example session:

```
$bin/gcc/debug/options_description --help
Usage: options_description [options]
Allowed options:
  --help                : produce help message
  --optimization arg    : optimization level
  -I [ --include-path ] arg : include path
  --input-file arg      : input file
$bin/gcc/debug/options_description
```

```
Optimization level is 10
$bin/gcc/debug/options_description --optimization 4 -I foo a.cpp
Include paths are: foo
Input files are: a.cpp
Optimization level is 4
```

Oops, there's a slight problem. It's still possible to specify the "--input-file" option, and usage message says so, which can be confusing for the user. It would be nice to hide this information, but let's wait for the next example.

Multiple Sources

It's quite likely that specifying all options to our compiler on the command line will annoy users. What if a user installs a new library and wants to always pass an additional command line element? What if he has made some choices which should be applied on every run? It's desirable to create a config file with common settings which will be used together with the command line.

Of course, there will be a need to combine the values from command line and config file. For example, the optimization level specified on the command line should override the value from the config file. On the other hand, include paths should be combined.

Let's see the code now. The complete program is in "examples/multiple_sources.cpp". The option definition has two interesting details. First, we declare several instances of the `options_description` class. The reason is that, in general, not all options are alike. Some options, like "input-file" above, should not be presented in an automatic help message. Some options make sense only in the config file. Finally, it's nice to have some structure in the help message, not just a long list of options. Let's declare several option groups:

```
// Declare a group of options that will be
// allowed only on command line
po::options_description generic("Generic options");
generic.add_options()
    ("version,v", "print version string")
    ("help", "produce help message")
    ;

// Declare a group of options that will be
// allowed both on command line and in
// config file
po::options_description config("Configuration");
config.add_options()
    ("optimization", po::value<int>(&opt)->default_value(10),
          "optimization level")
    ("include-path,I",
         po::value< vector<string> >()->composing(),
         "include path")
    ;

// Hidden options, will be allowed both on command line and
// in config file, but will not be shown to the user.
po::options_description hidden("Hidden options");
hidden.add_options()
    ("input-file", po::value< vector<string> >(), "input file")
    ;
```

Note the call to the `composing` method in the declaration of the "include-path" option. It tells the library that values from different sources should be composed together, as we'll see shortly.

The `add` method of the `options_description` class can be used to further group the options:

```
po::options_description cmdline_options;
cmdline_options.add(generic).add(config).add(hidden);

po::options_description config_file_options;
config_file_options.add(config).add(hidden);

po::options_description visible("Allowed options");
visible.add(generic).add(config);
```

The parsing and storing of values follows the usual pattern, except that we additionally call `parse_config_file`, and call the `store` function twice. But what happens if the same value is specified both on the command line and in config file? Usually, the value stored first is preferred. This is what happens for the "--optimization" option. For "composing" options, like "include-file", the values are merged.

Here's an example session:

```
$bin/gcc/debug/multiple_sources
Include paths are: /opt
Optimization level is 1
$bin/gcc/debug/multiple_sources --help
Allows options:

Generic options:
  -v [ --version ]       : print version string
  --help                 : produce help message

Configuration:
  --optimization n       : optimization level
  -I [ --include-path ] path : include path

$bin/gcc/debug/multiple_sources --optimization=4 -I foo a.cpp b.cpp
Include paths are: foo /opt
Input files are: a.cpp b.cpp
Optimization level is 4
```

The first invocation uses values from the configuration file. The second invocation also uses values from command line. As we see, the include paths on the command line and in the configuration file are merged, while optimization is taken from the command line.

Library Overview

In the tutorial section, we saw several examples of library usage. Here we will describe the overall library design including the primary components and their function.

The library has three main components:

- The options description component, which describes the allowed options and what to do with the values of the options.
- The parsers component, which uses this information to find option names and values in the input sources and return them.
- The storage component, which provides the interface to access the value of an option. It also converts the string representation of values that parsers return into desired C++ types.

To be a little more concrete, the `options_description` class is from the options description component, the `parse_command_line` function is from the parsers component, and the `variables_map` class is from the storage component.

In the tutorial we've learned how those components can be used by the `main` function to parse the command line and config file. Before going into the details of each component, a few notes about the world outside of `main`.

For that outside world, the storage component is the most important. It provides a class which stores all option values and that class can be freely passed around your program to modules which need access to the options. All the other components can be used only in the place where the actual parsing is the done. However, it might also make sense for the individual program modules to describe their options and pass them to the main module, which will merge all options. Of course, this is only important when the number of options is large and declaring them in one place becomes troublesome.

Options Description Component

The options description component has three main classes: `option_description`, `value_semantic` and `options_description`. The first two together describe a single option. The `option_description` class contains the option's name, description and a pointer to `value_semantic`, which, in turn, knows the type of the option's value and can parse the value, apply the default value, and so on. The `options_description` class is a container for instances of `option_description`.

For almost every library, those classes could be created in a conventional way: that is, you'd create new options using constructors and then call the `add` method of `options_description`. However, that's overly verbose for declaring 20 or 30 options. This concern led to creation of the syntax that you've already seen:

```
options_description desc;
desc.add_options()
    ("help", "produce help")
    ("optimization", value<int>()->default_value(10), "optimization level")
    ;
```

The call to the `value` function creates an instance of a class derived from the `value_semantic` class: `typed_value`. That class contains the code to parse values of a specific type, and contains a number of methods which can be called by the user to specify additional information. (This essentially emulates named parameters of the constructor.) Calls to `operator()` on the object returned by `add_options` forward arguments to the constructor of the `option_description` class and add the new instance.

Note that in addition to the `value`, library provides the `bool_switch` function, and user can write his own function which will return other subclasses of `value_semantic` with different behaviour. For the remainder of this section, we'll talk only about the `value` function.

The information about an option is divided into syntactic and semantic. Syntactic information includes the name of the option and the number of tokens which can be used to specify the value. This information is used by parsers to group tokens into (name, value) pairs, where value is just a vector of strings (`std::vector<std::string>`). The semantic layer is responsible for converting the value of the option into more usable C++ types.

This separation is an important part of library design. The parsers use only the syntactic layer, which takes away some of the freedom to use overly complex structures. For example, it's not easy to parse syntax like:

```
calc --expression=1 + 2/3
```

because it's not possible to parse

```
1 + 2/3
```

without knowing that it's a C expression. With a little help from the user the task becomes trivial, and the syntax clear:

```
calc --expression="1 + 2/3"
```

Syntactic Information

The syntactic information is provided by the `boost::program_options::options_description` class and some methods of the `boost::program_options::value_semantic` class and includes:

- name of the option, used to identify the option inside the program,
- description of the option, which can be presented to the user,
- the allowed number of source tokens that comprise options's value, which is used during parsing.

Consider the following example:

```
options_description desc;
desc.add_options()
    ("help", "produce help message")
    ("compression", value<string>(), "compression level")
    ("verbose", value<string>()->implicit(), "verbosity level")
    ("email", value<string>()->multitoken(), "email to send to")
    ;
```

For the first parameter, we specify only the name and the description. No value can be specified in the parsed source. For the first option, the user must specify a value, using a single token. For the third option, the user may either provide a single token for the value, or no token at all. For the last option, the value can span several tokens. For example, the following command line is OK:

```
test --help --compression 10 --verbose --email beadle@mars beadle2@mars
```

Description formatting

Sometimes the description can get rather long, for example, when several option's values need separate documentation. Below we describe some simple formatting mechanisms you can use.

The description string has one or more paragraphs, separated by the newline character ('\n'). When an option is output, the library will compute the indentation for options's description. Each of the paragraph is output as a separate line with that intentation. If a paragraph does not fit on one line it is spanned over multiple lines (which will have the same indentation).

You may specify additional indent for the first specified by inserting spaces at the beginning of a paragraph. For example:

```
options.add_options()
    ("help", "   A long help msg a long help msg a long help msg a long help
msg a long help msg a long help msg a long help msg a long help msg ")
    ;
```

will specify a four-space indent for the first line. The output will look like:

```
  --help                    A long help msg a long
                          help msg a long help msg
                          a long help msg a long
                          help msg a long help msg
                          a long help msg a long
                          help msg
```

For the case where line is wrapped, you can want an additional indent for wrapped text. This can be done by inserting a tabulator character ('\t') at the desired position. For example:

```
options.add_options()
      ("well_formated", "As you can see this is a very well formatted
option description.\n"
                        "You can do this for example:\n\n"
                        "Values:\n"
                        "  Value1: \tdoes this and that, bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla\n"
                        "  Value2: \tdoes something else, bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla\n\n"
                        "    This paragraph has a first line indent only,
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla");
```

will produce:

```
  --well_formated       As you can see this is a
                        very well formatted
                        option description.
                        You can do this for
                        example:

                        Values:
                          Value1: does this and
                                  that, bla bla
                                  bla bla bla bla
                                  bla bla bla bla
                                  bla bla bla bla
                                  bla
                          Value2: does something
                                  else, bla bla
                                  bla bla bla bla
                                  bla bla bla bla
                                  bla bla bla bla
                                  bla

                            This paragraph has a
                        first line indent only,
                        bla bla bla bla bla bla
```

```
                    bla bla bla bla bla bla
                    bla bla bla
```

The tab character is removed before output. Only one tabulator per paragraph is allowed, otherwisee an exception of type program_options::error is thrown. Finally, the tabulator is ignored if it's is not on the first line of the paragraph or is on the last possible position of the first line.

Semantic Information

The semantic information is completely provided by the `boost::program_options::value_semantic` class. For example:

```
options_description desc;
desc.add_options()
    ("compression", value<int>()->default_value(10), "compression level")
    ("email", value< vector<string> >()
        ->composing()->notifier(&your_function), "email")
    ;
```

These declarations specify that default value of the first option is 10, that the second option can appear several times and all instances should be merged, and that after parsing is done, the library will call function `&your_function`, passing the value of the "email" option as argument.

Positional Options

Our definition of option as (name, value) pairs is simple and useful, but in one special case of the command line, there's a problem. A command line can include a *positional option*, which does not specify any name at all, for example:

```
        archiver --compression=9 /etc/passwd
```

Here, the "/etc/passwd" element does not have any option name.

One solution is to ask the user to extract positional options himself and process them as he likes. However, there's a nicer approach -- provide a method to automatically assign the names for positional options, so that the above command line can be interpreted the same way as:

```
        archiver --compression=9 --input-file=/etc/passwd
```

The `positional_options_description` class allows the command line parser to assign the names. The class specifies how many positional options are allowed, and for each allowed option, specifies the name. For example:

```
positional_options_description pd; pd.add("input-file", 1);
```

specifies that for exactly one, first, positional option the name will be "input-file".

It's possible to specify that a number, or even all positional options, be given the same name.

```
positional_options_description pd;
pd.add("output-file", 2).add_optional("input-file", -1);
```

In the above example, the first two positional options will be associated with name "output-file", and any others with the name "input-file".

Parsers Component

The parsers component splits input sources into (name, value) pairs. Each parser looks for possible options and consults the options description component to determine if the option is known and how its value is specified. In the simplest case, the name is explicitly specified, which allows the library to decide if such option is known. If it is known, the `value_semantic` instance determines how the value is specified. (If it is not known, an exception is thrown.) Common cases are when the value is explicitly specified by the user, and when the value cannot be specified by the user, but the presence of the option implies some value (for example, `true`). So, the parser checks that the value is specified when needed and not specified when not needed, and returns new (name, value) pair.

To invoke a parser you typically call a function, passing the options description and command line or config file or something else. The results of parsing are returned as an instance of the `parsed_options` class. Typically, that object is passed directly to the storage component. However, it also can be used directly, or undergo some additional processing.

There are three exceptions to the above model -- all related to traditional usage of the command line. While they require some support from the options description component, the additional complexity is tolerable.

- The name specified on the command line may be different from the option name -- it's common to provide a "short option name" alias to a longer name. It's also common to allow an abbreviated name to be specified on the command line.
- Sometimes it's desirable to specify value as several tokens. For example, an option "--email-recipient" may be followed by several emails, each as a separate command line token. This behaviour is supported, though it can lead to parsing ambiguities and is not enabled by default.
- The command line may contain positional options -- elements which don't have any name. The command line parser provides a mechanism to guess names for such options, as we've seen in the tutorial.

Storage Component

The storage component is responsible for:

- Storing the final values of an option into a special class and in regular variables
- Handling priorities among different sources.
- Calling user-specified `notify` functions with the final values of options.

Let's consider an example:

```
variables_map vm;
store(parse_command_line(argc, argv, desc), vm);
store(parse_config_file("example.cfg", desc), vm);
notify(vm);
```

The `variables_map` class is used to store the option values. The two calls to the `store` function add values found on the command line and in the config file. Finally the call to the `notify` function runs the user-specified notify functions and stores the values into regular variables, if needed.

The priority is handled in a simple way: the `store` function will not change the value of an option if it's already assigned. In this case, if the command line specifies the value for an option, any value in the config file is ignored.

Warning

Don't forget to call the `notify` function after you've stored all parsed values.

Specific parsers

Environment variables

Environment variables are string variables which are available to all programs via the `getenv` function of C runtime library. The operating system allows to set initial values for a given user, and the values can be further changed on the command line. For example, on Windows one can use the `autoexec.bat` file or (on recent versions) the `Control Panel/System/Advanced/Environment Variables` dialog, and on Unix —, the `/etc/profile`, `~/profile` and `~/bash_profile` files. Because environment variables can be set for the entire system, they are particularly suitable for options which apply to all programs.

The environment variables can be parsed with the `parse_environment` function. The function have several overloaded versions. The first parameter is always an `options_description` instance, and the second specifies what variables must be processed, and what option names must correspond to it. To describe the second parameter we need to consider naming conventions for environment variables.

If you have an option that should be specified via environment variable, you need make up the variable's name. To avoid name clashes, we suggest that you use a sufficiently unique prefix for environment variables. Also, while option names are most likely in lower case, environment variables conventionally use upper case. So, for an option name `proxy` the environment variable might be called `BOOST_PROXY`. During parsing, we need to perform reverse conversion of the names. This is accomplished by passing the choosen prefix as the second parameter of the `parse_environment` function. Say, if you pass `BOOST_` as the prefix, and there are two variables, `CVSROOT` and `BOOST_PROXY`, the first variable will be ignored, and the second one will be converted to option `proxy`.

The above logic is sufficient in many cases, but it is also possible to pass, as the second parameter of the `parse_environment` function, any function taking a `std::string` and returning `std::string`. That function will be called for each environment variable and should return either the name of the option, or empty string if the variable should be ignored.

Annotated List of Symbols

The following table describes all the important symbols in the library, for quick access.

Symbol	Description
Options description component	
`options_description`	describes a number of options
`value`	defines the option's value
Parsers component	
`parse_command_line`	parses command line
`parse_config_file`	parses config file

Symbol	Description
`parse_environment`	parses environment
Storage component	
`variables_map`	storage for option values

How To

This section describes how the library can be used in specific situations.

Non-conventional Syntax

Sometimes, standard command line syntaxes are not enough. For example, the gcc compiler has "-frtti" and -fno-rtti" options, and this syntax is not directly supported.

For such cases, the library allows the user to provide an *additional parser* -- a function which will be called on each command line element, before any processing by the library. If the additional parser recognises the syntax, it returns the option name and value, which are used directly. The above example can be handled by the following code:

```
pair<string, string> reg_foo(const string& s)
{
    if (s.find("-f") == 0) {
        if (s.substr(2, 3) == "no-")
            return make_pair(s.substr(5), string("false"));
        else
            return make_pair(s.substr(2), string("true"));
    } else {
        return make_pair(string(), string());
    }
}
```

Here's the definition of the additional parser. When parsing the command line, we pass the additional parser:

```
store(command_line_parser(ac, av).options(desc).extra_parser(reg_foo)
        .run(), vm);
```

The complete example can be found in the "example/custom_syntax.cpp" file.

Response Files

Some operating system have very low limits of the command line length. The common way to work around those limitations is using *response files*. A response file is just a configuration file which uses the same syntax as the command line. If the command line specifies a name of response file to use, it's loaded and parsed in addition to the command line. The library does not provide direct support for response files, so you'll need to write some extra code.

First, you need to define an option for the response file:

```
("response-file", value<string>(),
     "can be specified with '@name', too")
```

Second, you'll need an additional parser to support the standard syntax for specifying response files: "@file":

```
pair<string, string> at_option_parser(string const&s)
{
    if ('@' == s[0])
        return std::make_pair(string("response-file"), s.substr(1));
    else
        return pair<string, string>();
}
```

Finally, when the "response-file" option is found, you'll have to load that file and pass it to the command line parser. This part is the hardest. We'll use the Boost.Tokenizer library, which works but has some limitations. You might also consider Boost.StringAlgo. The code is:

```
if (vm.count("response-file")) {
     // Load the file and tokenize it
     ifstream ifs(vm["response-file"].as<string>().c_str());
     if (!ifs) {
         cout << "Could no open the response file\n";
         return 1;
     }
     // Read the whole file into a string
     stringstream ss;
     ss << ifs.rdbuf();
     // Split the file content
     char_separator<char> sep(" \n\r");
     tokenizer<char_separator<char> > tok(ss.str(), sep);
     vector<string> args;
     copy(tok.begin(), tok.end(), back_inserter(args));
     // Parse the file and store the options
     store(command_line_parser(args).options(desc).run(), vm);
}
```

The complete example can be found in the "example/response_file.cpp" file.

Winmain Command Line

On the Windows operating system, GUI applications receive the command line as a single string, not split into elements. For that reason, the command line parser cannot be used directly. At least on some compilers, it is possible to obtain the split command line, but it's not clear if all compilers support the same mechanism on all versions of the operating system. The `split_winmain` function is a portable mechanism provided by the library.

Here's an example of use:

```
vector<string> args = split_winmain(lpCmdLine);
store(command_line_parser(args).options(desc).run(), vm);
```

The function is an overload for `wchar_t` strings, so can also be used in Unicode applications.

Option Groups and Hidden Options

Having a single instance of the `options_description` class with all the program's options can be problematic:

- Some options make sense only for specific source, for example, configuration files.
- The user would prefer some structure in the generated help message.
- Some options shouldn't appear in the generated help message at all.

To solve the above issues, the library allows a programmer to create several instances of the `options_description` class, which can be merged in different combinations. The following example will define three groups of options: command line specific, and two options group for specific program modules, only one of which is shown in the generated help message.

Each group is defined using standard syntax. However, you should use reasonable names for each `options_description` instance:

```
options_description general("General options");
general.add_options()
    ("help", "produce a help message")
    ("help-module", value<string>()->implicit(),
        "produce a help for a given module")
    ("version", "output the version number")
    ;

options_description gui("GUI options");
gui.add_options()
    ("display", value<string>(), "display to use")
    ;

options_description backend("Backend options");
backend.add_options()
    ("num-threads", value<int>(), "the initial number of threads")
    ;
```

After declaring options groups, we merge them in two combinations. The first will include all options and be used for parsing. The second will be used for the "--help" option.

```
// Declare an options description instance which will include
// all the options
options_description all("Allowed options");
all.add(general).add(gui).add(backend);

// Declare an options description instance which will be shown
// to the user
options_description visible("Allowed options");
visible.add(general).add(gui);
```

What is left is to parse and handle the options:

```
variables_map vm;
store(parse_command_line(ac, av, all), vm);

if (vm.count("help"))
{
```

```
    cout << visible;
    return 0;
}
if (vm.count("help-module")) {
    const string& s = vm["help-module"].as<string>();
    if (s == "gui") {
        cout << gui;
    } else if (s == "backend") {
        cout << backend;
    } else {
        cout << "Unknown module '"
             << s << "' in the --help-module option\n";
        return 1;
    }
    return 0;
}
if (vm.count("num-threads")) {
    cout << "The 'num-threads' options was set to "
         << vm["num-threads"].as<int>() << "\n";
}
```

When parsing the command line, all options are allowed. The "--help" message, however, does not include the "Backend options" group -- the options in that group are hidden. The user can explicitly force the display of that options group by passing "--help-module backend" option. The complete example can be found in the "example/option_groups.cpp" file.

Custom Validators

By default, the conversion of option's value from string into C++ type is done using iostreams, which sometimes is not convenient. The library allows the user to customize the conversion for specific classes. In order to do so, the user should provide suitable overload of the `validate` function.

Let's first define a simple class:

```
struct magic_number {
public:
    magic_number(int n) : n(n) {}
    int n;
};
```

and then overload the `validate` function:

```
void validate(boost::any& v,
              const std::vector<std::string>& values,
              magic_number* target_type, int)
{
    static regex r("\\d\\d\\d-(\\d\\d\\d)");

    using namespace boost::program_options;

    // Make sure no previous assignment to 'a' was made.
    validators::check_first_occurence(v);
    // Extract the first string from 'values'. If there is more than
    // one string, it's an error, and exception will be thrown.
    const string& s = validators::get_single_string(values);

    // Do regex match and convert the interesting part to
    // int.
    smatch match;
    if (regex_match(s, match, r)) {
```

```
        v = any(magic_number(lexical_cast<int>(match[1])));
    } else {
        throw validation_error("invalid value");
    }
}
```

The function takes four parameters. The first is the storage for the value, and in this case is either empty or contains an instance of the `magic_number` class. The second is the list of strings found in the next occurrence of the option. The remaining two parameters are needed to workaround the lack of partial template specialization and partial function template ordering on some compilers.

The function first checks that we don't try to assign to the same option twice. Then it checks that only a single string was passed in. Next the string is verified with the help of the Boost.Regex library. If that test is passed, the parsed value is stored into the `v` variable.

The complete example can be found in the "example/regex.cpp" file.

Unicode Support

To use the library with Unicode, you'd need to:

- Use Unicode-aware parsers for Unicode input
- Require Unicode support for options which need it

Most of the parsers have Unicode versions. For example, the `parse_command_line` function has an overload which takes `wchar_t` strings, instead of ordinary `char`.

Even if some of the parsers are Unicode-aware, it does not mean you need to change definition of all the options. In fact, for many options, like integer ones, it makes no sense. To make use of Unicode you'll need *some* Unicode-aware options. They are different from ordinary options in that they accept `wstring` input, and process it using wide character streams. Creating an Unicode-aware option is easy: just use the the `wvalue` function instead of the regular `value`.

When an ascii parser passes data to an ascii option, or a Unicode parser passes data to a Unicode option, the data are not changed at all. So, the ascii option will see a string in local 8-bit encoding, and the Unicode option will see whatever string was passed as the Unicode input.

What happens when Unicode data is passed to an ascii option, and vice versa? The library automatically performs the conversion from Unicode to local 8-bit encoding. For example, if command line is in ascii, but you use `wstring` options, then the ascii input will be converted into Unicode.

To perform the conversion, the library uses the `codecvt<wchar_t, char>` locale facet from the global locale. If you want to work with strings that use local 8-bit encoding (as opposed to 7-bit ascii subset), your application should start with:

```
locale::global(locale(""));
```

which would set up the conversion facet according to the user's selected locale.

It's wise to check the status of the C++ locale support on your implementation, though. The quick test involves three steps:

1. Go the the "test" directory and build the "test_convert" binary.

2. Set some non-ascii locale in the environmemt. On Linux, one can run, for example:

```
$ export LC_CTYPE=ru_RU.KOI8-R
```

3. Run the "test_convert" binary with any non-ascii string in the selected encoding as its parameter. If you see a list of Unicode codepoints, everything's OK. Otherwise, locale support on your system might be broken.

Allowing Unknown Options

Usually, the library throws an exception on unknown option names. This behaviour can be changed. For example, only some part of your application uses Program_options, and you wish to pass unrecognized options to another part of the program, or even to another application.

To allow unregistered options on the command line, you need to use the `basic_command_line_parser` class for parsing (not `parse_command_line`) and call the `allow_unregistered` method of that class:

```
parsed_options parsed =
    command_line_parser(argv, argc).options(desc).allow_unregistered().run();
```

For each token that looks like an option, but does not have a known name, an instance of `basic_option` will be added to the result. The `string_key` and `value` fields of the instance will contain results of syntactic parsing of the token, the `unregistered` field will be set to `true`, and the `original_tokens` field will contain the token as it appeared on the command line.

If you want to pass the unrecognized options further, the `collect_unrecognized` function can be used. The function will collect original tokens for all unrecognized values, and optionally, all found positional options. Say, if your code handles a few options, but does not handles positional options at all, you can use the function like this:

```
vector<string> to_pass_further = collect_arguments(parsed.option, include_position
```

Design Discussion

This section focuses on some of the design questions.

Unicode Support

Unicode support was one of the features specifically requested during the formal review. Throughout this document "Unicode support" is a synonym for "wchar_t" support, assuming that "wchar_t" always uses Unicode encoding. Also, when talking about "ascii" (in lowercase) we'll not mean strict 7-bit ASCII encoding, but rather "char" strings in local 8-bit encoding.

Generally, "Unicode support" can mean many things, but for the program_options library it means that:

- Each parser should accept either `char*` or `wchar_t*`, correctly split the input into option names

and option values and return the data.

- For each option, it should be possible to specify whether the conversion from string to value uses ascii or Unicode.
- The library guarantees that:
  - ascii input is passed to an ascii value without change
  - Unicode input is passed to a Unicode value without change
  - ascii input passed to a Unicode value, and Unicode input passed to an ascii value will be converted using a codecvt facet (which may be specified by the user(which can be specified by the user)

The important point is that it's possible to have some "ascii options" together with "Unicode options". There are two reasons for this. First, for a given type you might not have the code to extract the value from Unicode string and it's not good to require that such code be written. Second, imagine a reusable library which has some options and exposes options description in its interface. If *all* options are either ascii or Unicode, and the library does not use any Unicode strings, then the author will likely to use ascii options, which would make the library unusable inside Unicode applications. Essentially, it would be necessary to provide two versions of the library -- ascii and Unicode.

Another important point is that ascii strings are passed though without modification. In other words, it's not possible to just convert ascii to Unicode and process the Unicode further. The problem is that the default conversion mechanism -- the `codecvt` facet -- might not work with 8-bit input without additional setup.

The Unicode support outlined above is not complete. For example, we don't plan allow Unicode in option names. Unicode support is hard and requires a Boost-wide solution. Even comparing two arbitrary Unicode strings is non-trivial. Finally, using Unicode in option names is related to internationalization, which has it's own complexities. E.g. if option names depend on current locale, then all program parts and other parts which use the name must be internationalized too.

The primary question in implementing the Unicode support is whether to use templates and `std::basic_string` or to use some internal encoding and convert between internal and external encodings on the interface boundaries.

The choice, mostly, is between code size and execution speed. A templated solution would either link library code into every application that uses the library (thereby making shared library impossible), or provide explicit instantiations in the shared library (increasing its size). The solution based on internal encoding would necessarily make conversions in a number of places and will be somewhat slower. Since speed is generally not an issue for this library, the second solution looks more attractive, but we'll take a closer look at individual components.

For the parsers component, we have three choices:

- Use a fully templated implementation: given a string of a certain type, a parser will return a `parsed_options` instance with strings of the same type (i.e. the `parsed_options` class will be templated).
- Use internal encoding: same as above, but strings will be converted to and from the internal encoding.
- Use and partly expose the internal encoding: same as above, but the strings in the `parsed_options` instance will be in the internal encoding. This might avoid a conversion if `parsed_options` instance is passed directly to other components, but can be also dangerous or

confusing for a user.

The second solution appears to be the best -- it does not increase the code size much and is cleaner than the third. To avoid extra conversions, the Unicode version of `parsed_options` can also store strings in internal encoding.

For the options descriptions component, we don't have much choice. Since it's not desirable to have either all options use ascii or all of them use Unicode, but rather have some ascii and some Unicode options, the interface of the `value_semantic` must work with both. The only way is to pass an additional flag telling if strings use ascii or internal encoding. The instance of `value_semantic` can then convert into some other encoding if needed.

For the storage component, the only affected function is `store`. For Unicode input, the `store` function should convert the value to the internal encoding. It should also inform the `value_semantic` class about the used encoding.

Finally, what internal encoding should we use? The alternatives are: `std::wstring` (using UCS-4 encoding) and `std::string` (using UTF-8 encoding). The difference between alternatives is:

- Speed: UTF-8 is a bit slower
- Space: UTF-8 takes less space when input is ascii
- Code size: UTF-8 requires additional conversion code. However, it allows one to use existing parsers without converting them to `std::wstring` and such conversion is likely to create a number of new instantiations.

There's no clear leader, but the last point seems important, so UTF-8 will be used.

Choosing the UTF-8 encoding allows the use of existing parsers, because 7-bit ascii characters retain their values in UTF-8, so searching for 7-bit strings is simple. However, there are two subtle issues:

- We need to assume the character literals use ascii encoding and that inputs use Unicode encoding.
- A Unicode character (say '=') can be followed by 'composing character' and the combination is not the same as just '=', so a simple search for '=' might find the wrong character.

Neither of these issues appear to be critical in practice, since ascii is almost universal encoding and since composing characters following '=' (and other characters with special meaning to the library) are not likely to appear.

Acknowledgements

I'm very gratefull to all the people who helped with the development, by discussion, fixes, and as users. It was pleasant to see all that involvement, which made the library much better than it would be otherwise.

In the early stages, the library was affected by discussions with Gennadiy Rozental, William Kempf and Alexander Okhotin.

Hartmut Kaiser was the first person to try the library on his project and send a number of suggestions and fixes.

The formal review lead to numerous comments and enhancements. Pavol Droba helped with the option description semantic. Gennadiy Rozental has criticised many aspects of the library which caused various

simplifications. Pavel Vozenilek did carefull review of the implementation. A number of comments were made by:

- David Abrahams
- Neal D. Becker
- Misha Bergal
- James Curran
- Carl Daniel
- Beman Dawes
- Tanton Gibbs
- Holger Grund
- Hartmut Kaiser
- Petr Kocmid
- Baptiste Lepilleur
- Marcelo E. Magallon
- Chuck Messenger
- John Torjo
- Matthias Troyer

Doug Gregor and Reece Dunn helped to resolve the issues with Boostbook version of the documentation.

Even after review, a number of people have helped with further development:

- Rob Lievaart
- Thorsten Ottosen
- Joseph Wu
- Ferdinand Prantl
- Miro Jurisic
- John Maddock
- Janusz Piwowarski
- Charles Brockman
- Jonathan Wakely

Reference

Header <boost/program_options/cmdline.hpp>

```
namespace boost {
  namespace program_options {
    namespace command_line_style {
      enum style_t;
    }
  }
}
```

Type style_t

Type style_t --

```
enum style_t { allow_long =  1, allow_short =  allow_long << 1, allow_dash_for_sho
               allow_slash_for_short =  allow_dash_for_short << 1, long_allow_adja
               short_allow_adjacent =  long_allow_next << 1, short_allow_next =  s
               allow_guessing =  allow_sticky << 1, case_insensitive =  allow_gues
               unix_style =  (allow_short | short_allow_adjacent | short_allow_nex
                      | allow_long | long_allow_adjacent | long_allow_next
                      | allow_sticky | allow_guessing
                      | allow_dash_for_short), default_style =  unix_style };
```

Header <boost/program_options/environment_iterator.hpp>

```
namespace boost {
  class environment_iterator;
}
```

Class environment_iterator

Class environment_iterator --

```
class environment_iterator : public boost::eof_iterator< environment_iterator, std
{
public:
  // construct/copy/destruct
  environment_iterator(char **);
  environment_iterator();

  // public member functions
  void get() ;
};
```

Description

environment_iterator construct/copy/destruct

1. ```
   environment_iterator(char ** environment);
   ```

2. ```
   environment_iterator();
   ```

environment_iterator public member functions

1. ```
   void get() ;
   ```

Header <boost/program_options/eof_iterator.hpp>

```
namespace boost {
  template<typename Derived, typename ValueType> class eof_iterator;
}
```
```

Class template eof_iterator

Class template eof_iterator --

```
template<typename Derived, typename ValueType>
class eof_iterator {
public:
  // construct/copy/destruct
  eof_iterator();

  // public member functions

  // protected member functions
  ValueType & value() ;
  void found_eof() ;

  // private member functions
  void increment() ;
  bool equal(const eof_iterator &) const;
  const ValueType & dereference() const;
};
```

Description

The 'eof_iterator' class is useful for constructing forward iterators in cases where iterator extract data from some source and it's easy to detect 'eof' -- i.e. the situation where there's no data. One apparent example is reading lines from a file.

Implementing such iterators using 'iterator_facade' directly would require to create class with three core operation, a couple of constructors. When using 'eof_iterator', the derived class should define only one method to get new value, plus a couple of constructors.

The basic idea is that iterator has 'eof' bit. Two iterators are equal only if both have their 'eof' bits set. The 'get' method either obtains the new value or sets the 'eof' bit.

Specifically, derived class should define:

1. A default constructor, which creates iterator with 'eof' bit set. The constructor body should call 'found_eof' method defined here. 2. Some other constructor. It should initialize some 'data pointer' used in iterator operation and then call 'get'. 3. The 'get' method. It should operate this way:

- look at some 'data pointer' to see if new element is available; if not, it should call 'found_eof'.
- extract new element and store it at location returned by the 'value' method.
- advance the data pointer.

Essentially, the 'get' method has the functionality of both 'increment' and 'dereference'. It's very good for the cases where data extraction implicitly moves data pointer, like for stream operation.

`eof_iterator` construct/copy/destruct

1.
```
eof_iterator();
```

`eof_iterator` public member functions

`eof_iterator` protected member functions

1. ```
   ValueType & value() ;
   ```

   Returns the reference which should be used by derived class to store the next value.

2. ```
   void found_eof() ;
   ```

   Should be called by derived class to indicate that it can't produce next element.

`eof_iterator` private member functions

1. ```
   void increment() ;
   ```

2. ```
   bool equal(const eof_iterator & other) const;
   ```

3. ```
   const ValueType & dereference() const;
   ```

Header <boost/program_options/errors.hpp>

```
namespace boost {
  namespace program_options {
    class error;
    class invalid_syntax;
    class unknown_option;
    class ambiguous_option;
    class multiple_values;
    class multiple_occurrences;
    class validation_error;
    class invalid_option_value;
    class too_many_positional_options_error;
    class too_few_positional_options_error;
    class invalid_command_line_syntax;
    class invalid_command_line_style;
  }
}
```
```

Class error

Class error --

```
class error {
public:
  // construct/copy/destruct
  error(const std::string &);

  // public member functions
};
```

Description

Base class for all errors in the library.

`error` construct/copy/destruct

1.
```
error(const std::string & what);
```

`error` public member functions

Class invalid_syntax

Class invalid_syntax --

```
class invalid_syntax : public boost::program_options::error {
public:
  // construct/copy/destruct
  invalid_syntax(const std::string &, const std::string &);
  ~invalid_syntax();

  // public member functions

  std::string tokens;
  std::string msg;
};
```

Description

`invalid_syntax` construct/copy/destruct

1. ```
   invalid_syntax(const std::string & tokens, const std::string & msg);
   ```

2. ```
   ~invalid_syntax();
   ```

`invalid_syntax` public member functions

Class unknown_option

Class unknown_option --

```
class unknown_option : public boost::program_options::error {
public:
  // construct/copy/destruct
  unknown_option(const std::string &);

  // public member functions
};
```

Description

Class thrown when option name is not recognized.

unknown_option construct/copy/destruct

1.
```
unknown_option(const std::string & name);
```

unknown_option public member functions

Class ambiguous_option

Class ambiguous_option --

```
class ambiguous_option : public boost::program_options::error {
public:
  // construct/copy/destruct
  ambiguous_option(const std::string &, const std::vector< std::string > &);
  ~ambiguous_option();

  // public member functions

  std::vector< std::string > alternatives;
};
```

Description

Class thrown when there's ambiguity amoung several possible options.

ambiguous_option construct/copy/destruct

1.
```
ambiguous_option(const std::string & name,
                 const std::vector< std::string > & alternatives);
```

2.
```
~ambiguous_option();
```

ambiguous_option public member functions

Class multiple_values

Class multiple_values --

```
class multiple_values : public boost::program_options::error {
public:
  // construct/copy/destruct
  multiple_values(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are several option values, but user called a method which cannot return them all.

`multiple_values` construct/copy/destruct

1.
```
multiple_values(const std::string & what);
```

`multiple_values` public member functions

Class multiple_occurrences

Class multiple_occurrences --

```
class multiple_occurrences : public boost::program_options::error {
public:
  // construct/copy/destruct
  multiple_occurrences(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are several occurrences of an option, but user called a method which cannot return them all.

multiple_occurrences construct/copy/destruct

1. ```
   multiple_occurrences(const std::string & what);
   ```

multiple_occurrences public member functions
   ```

Class validation_error

Class validation_error --

```
class validation_error : public boost::program_options::error {
public:
  // construct/copy/destruct
  validation_error(const std::string &);
  ~validation_error();

  // public member functions
  void set_option_name(const std::string &) ;

  // private member functions
  const char * what() const;
};
```

Description

Class thrown when value of option is incorrect.

`validation_error` construct/copy/destruct

1. `validation_error(const std::string & what);`

2. `~validation_error();`

`validation_error` public member functions

1. `void set_option_name(const std::string & option) ;`

`validation_error` private member functions

1. `const char * what() const;`

Class invalid_option_value

Class invalid_option_value --

```
class invalid_option_value
  :  : public boost::program_options::validation_error
{
public:
  // construct/copy/destruct
  invalid_option_value(const std::string &);
  invalid_option_value(const std::wstring &);

  // public member functions
};
```

Description

`invalid_option_value` construct/copy/destruct

1. ```
   invalid_option_value(const std::string & value);
   ```

2. ```
   invalid_option_value(const std::wstring & value);
   ```

`invalid_option_value` public member functions

Class too_many_positional_options_error

Class too_many_positional_options_error --

```
class too_many_positional_options_error
  :  : public boost::program_options::error
{
public:
  // construct/copy/destruct
  too_many_positional_options_error(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are too many positional options.

too_many_positional_options_error construct/copy/destruct

1.
```
too_many_positional_options_error(const std::string & what);
```

too_many_positional_options_error public member functions

Class too_few_positional_options_error

Class too_few_positional_options_error --

```
class too_few_positional_options_error
  :  : public boost::program_options::error
{
public:
  // construct/copy/destruct
  too_few_positional_options_error(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are too few positional options.

too_few_positional_options_error construct/copy/destruct

1.
```
too_few_positional_options_error(const std::string & what);
```

too_few_positional_options_error public member functions

Class invalid_command_line_syntax

Class invalid_command_line_syntax --

```
class invalid_command_line_syntax {
public:
  // construct/copy/destruct
  invalid_command_line_syntax(const std::string &, kind_t);

  // public member functions
  kind_t kind() const;

  // protected static functions
  std::string error_message(kind_t) ;
};
```

Description

`invalid_command_line_syntax` construct/copy/destruct

1. ```
   invalid_command_line_syntax(const std::string & tokens, kind_t kind);
   ```

`invalid_command_line_syntax` public member functions

1. ```
   kind_t kind() const;
   ```

`invalid_command_line_syntax` protected static functions

1. ```
   std::string error_message(kind_t kind) ;
   ```
   ```

Class invalid_command_line_style

Class invalid_command_line_style --

```
class invalid_command_line_style : public boost::program_options::error {
public:
  // construct/copy/destruct
  invalid_command_line_style(const std::string &);

  // public member functions
};
```

Description

`invalid_command_line_style` construct/copy/destruct

1.
```
invalid_command_line_style(const std::string & msg);
```

`invalid_command_line_style` public member functions

Header <boost/program_options/option.hpp>

```
namespace boost {
  namespace program_options {
    template<typename charT> class basic_option;

    typedef basic_option< char > option;
    typedef basic_option< wchar_t > woption;
  }
}
```

Class template basic_option

Class template basic_option --

```
template<typename charT>
class basic_option {
public:
  // construct/copy/destruct
  basic_option();
  basic_option(const std::string &, const std::vector< std::string > &);

  // public member functions

  std::string string_key;
  int position_key;
  std::vector< std::basic_string< charT > > value;
  std::vector< std::basic_string< charT > > original_tokens;
  bool unregistered;
};
```

Description

Option found in input source. Contains a key and a value. The key, in turn, can be a string (name of an option), or an integer (position in input source) -- in case no name is specified. The latter is only possible for command line. The template parameter specifies the type of char used for storing the option's value.

`basic_option` construct/copy/destruct

1.
```
basic_option();
```

2.
```
basic_option(const std::string & string_key,
             const std::vector< std::string > & value);
```

`basic_option` public member functions

Header <boost/program_options/options_description.hpp>

```
namespace boost {
  namespace program_options {
    class option_description;
    class options_description_easy_init;
    class options_description;
    class duplicate_option_error;
  }
}
```

Class option_description

Class option_description --

```
class option_description {
public:
  // construct/copy/destruct
  option_description();
  option_description(const char *, const value_semantic *);
  option_description(const char *, const value_semantic *, const char *);
  ~option_description();

  // public member functions
  bool match(const std::string &, bool) const;
  const std::string & key(const std::string &) const;
  const std::string & long_name() const;
  const std::string & description() const;
  shared_ptr< const value_semantic > semantic() const;
  std::string format_name() const;
  std::string format_parameter() const;

  // private member functions
  option_description & set_name(const char *) ;
};
```

Description

Describes one possible command line/config file option. There are two kinds of properties of an option. First describe it syntactically and are used only to validate input. Second affect interpretation of the option, for example default value for it or function that should be called when the value is finally known. Routines which perform parsing never use second kind of properties -- they are side effect free.

options_description

`option_description` construct/copy/destruct

1. ```
   option_description();
   ```

2. ```
   option_description(const char * name, const value_semantic * s);
   ```

   Initializes the object with the passed data.

   Note: it would be nice to make the second parameter auto_ptr, to explicitly pass ownership. Unfortunately, it's often needed to create objects of types derived from 'value_semantic': options_description d; d.add_options()("a", parameter<int>("n")->default_value(1)); Here, the static type returned by 'parameter' should be derived from value_semantic.

   Alas, derived->base conversion for auto_ptr does not really work, see http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2000/n1232.pdf http://std.dkuug.dk/jtc1/sc22/wg21/docs/cwg_defects.html#84

   So, we have to use plain old pointers. Besides, users are not expected to use the constructor dir-

ectly.

The 'name' parameter is interpreted by the following rules:

- if there's no "," character in 'name', it specifies long name
- otherwise, the part before "," specifies long name and the part after -- long name.

3.
```
option_description(const char * name, const value_semantic * s,
                   const char * description);
```

Initializes the class with the passed data.

4.
```
~option_description();
```

`option_description` public member functions

1.
```
bool match(const std::string & option, bool approx) const;
```

Given 'option', specified in the input source, return 'true' is 'option' specifies *this.

2.
```
const std::string & key(const std::string & option) const;
```

Return the key that should identify the option, in particular in the variables_map class. The 'option' parameter is the option spelling from the input source. If option name contains '*', returns 'option'. If long name was specified, it's the long name, otherwise it's a short name with prepended '-'.

3.
```
const std::string & long_name() const;
```

4.
```
const std::string & description() const;
```

5.
```
shared_ptr< const value_semantic > semantic() const;
```

6.
```
std::string format_name() const;
```

7.
```
std::string format_parameter() const;
```

Return the parameter name and properties, formatted suitably for usage message.

`option_description` private member functions

1.
```
option_description & set_name(const char * name) ;
```

Class options_description_easy_init

Class options_description_easy_init --

```
class options_description_easy_init {
public:
  // construct/copy/destruct
  options_description_easy_init(options_description *);

  // public member functions
  options_description_easy_init & operator()(const char *, const char *) ;
  options_description_easy_init &
  operator()(const char *, const value_semantic *) ;
  options_description_easy_init &
  operator()(const char *, const value_semantic *, const char *) ;
};
```

Description

Class which provides convenient creation syntax to option_description.

`options_description_easy_init` construct/copy/destruct

1.
```
options_description_easy_init(options_description * owner);
```

`options_description_easy_init` public member functions

1.
```
options_description_easy_init &
operator()(const char * name, const char * description) ;
```

2.
```
options_description_easy_init &
operator()(const char * name, const value_semantic * s) ;
```

3.
```
options_description_easy_init &
operator()(const char * name, const value_semantic * s,
           const char * description) ;
```

Class options_description

Class options_description --

```
class options_description {
public:
  // construct/copy/destruct
  options_description(unsigned = m_default_line_length);
  options_description(const std::string &, unsigned = m_default_line_length);

  // public member functions
  void add(shared_ptr< option_description >) ;
  options_description & add(const options_description &) ;
  options_description_easy_init add_options() ;
  const option_description & find(const std::string &, bool) const;
  const option_description * find_nothrow(const std::string &, bool) const;
  const std::vector< shared_ptr< option_description > > & options() const;
  void print(std::ostream &) const;

  static const unsigned m_default_line_length;
};
```

Description

A set of option descriptions. This provides convenient interface for adding new option (the add_options) method, and facilities to search for options by name.

See here for option adding interface discussion.

option_description

`options_description` construct/copy/destruct

1.
```
options_description(unsigned line_length = m_default_line_length);
```

Creates the instance.

2.
```
options_description(const std::string & caption,
                    unsigned line_length = m_default_line_length);
```

Creates the instance. The 'caption' parameter gives the name of this 'options_description' instance. Primarily useful for output.

`options_description` public member functions

1.
```
void add(shared_ptr< option_description > desc) ;
```

Adds new variable description. Throws duplicate_variable_error if either short or long name matches that of already present one.

2.
```
options_description & add(const options_description & desc) ;
```

Adds a group of option description. This has the same effect as adding all option_descriptions in 'desc' individually, except that output operator will show a separate group. Returns *this.

3.
```
options_description_easy_init add_options() ;
```

Returns an object of implementation-defined type suitable for adding options to options_description. The returned object will have overloaded operator() with parameter type matching 'option_description' constructors. Calling the operator will create new option_description instance and add it.

4.
```
const option_description & find(const std::string & name, bool approx) const;
```

5.
```
const option_description *
find_nothrow(const std::string & name, bool approx) const;
```

6.
```
const std::vector< shared_ptr< option_description > > & options() const;
```

7.
```
void print(std::ostream & os) const;
```

Output 'desc' to the specified stream, calling 'f' to output each option_description element.

Class duplicate_option_error

Class duplicate_option_error --

```
class duplicate_option_error : public boost::program_options::error {
public:
  // construct/copy/destruct
  duplicate_option_error(const std::string &);

  // public member functions
};
```

Description

Class thrown when duplicate option description is found.

duplicate_option_error construct/copy/destruct

1.
```
duplicate_option_error(const std::string & what);
```

duplicate_option_error public member functions

Header <boost/program_options/parsers.hpp>

```
namespace boost {
  namespace program_options {
    template<typename charT> class basic_parsed_options;

    template<> class basic_parsed_options<wchar_t>;

    template<typename charT> class basic_command_line_parser;

    typedef basic_parsed_options< char > parsed_options;
    typedef basic_parsed_options< wchar_t > wparsed_options;
    typedef function1< std::pair< std::string, std::string >, const std::string &
    typedef basic_command_line_parser< char > command_line_parser;
    typedef basic_command_line_parser< wchar_t > wcommand_line_parser;

    enum collect_unrecognized_mode;
    template<typename charT>
      basic_parsed_options< charT >
      parse_command_line(int, charT *, const options_description &, int = 0,
                         function1< std::pair< std::string, std::string >, const s
    template<typename charT>
      BOOST_PROGRAM_OPTIONS_DECL basic_parsed_options< charT >
      parse_config_file(std::basic_istream< charT > &,
                        const options_description &);
    template<typename charT>
      std::vector< std::basic_string< charT > >
      collect_unrecognized(const std::vector< basic_option< charT > > &,
                           enum collect_unrecognized_mode);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
```

```
    parse_environment(const options_description &,
                      const function1< std::string, std::string > &);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
    parse_environment(const options_description &, const std::string &);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
    parse_environment(const options_description &, const char *);
  }
}
```

Class template basic_parsed_options

Class template basic_parsed_options --

```
template<typename charT>
class basic_parsed_options {
public:
  // construct/copy/destruct
  basic_parsed_options(const options_description *);

  // public member functions

  std::vector< basic_option< charT > > options;
  const options_description * description;
};
```

Description

Results of parsing an input source. The primary use of this class is passing information from parsers component to value storage component. This class does not makes much sense itself.

basic_parsed_options construct/copy/destruct

1.
```
basic_parsed_options(const options_description * description);
```

basic_parsed_options public member functions

Specializations

- Class basic_parsed_options<wchar_t>

Class basic_parsed_options<wchar_t>

Class basic_parsed_options<wchar_t> --

```
class basic_parsed_options<wchar_t> {
public:

  // public member functions
   basic_parsed_options(const basic_parsed_options< char > &) ;

  std::vector< basic_option< wchar_t > > options;
  const options_description * description;
  basic_parsed_options< char > utf8_encoded_options;
};
```

Description

Specialization of basic_parsed_options which:

- provides convenient conversion from basic_parsed_options<char>
- stores the passed char-based options for later use.

`basic_parsed_options` public member functions

1. ```
   basic_parsed_options(const basic_parsed_options< char > & po) ;
   ```

   Constructs wrapped options from options in UTF8 encoding.
   ```

Class template basic_command_line_parser

Class template basic_command_line_parser --

```
template<typename charT>
class basic_command_line_parser {
public:
  // construct/copy/destruct
  basic_command_line_parser(const std::vector< std::basic_string< charT > > &);
  basic_command_line_parser(int, charT *);

  // public member functions
  basic_command_line_parser & options(const options_description &) ;
  basic_command_line_parser &
  positional(const positional_options_description &) ;
  basic_command_line_parser & style(int) ;
  basic_command_line_parser & extra_parser(ext_parser) ;
  basic_parsed_options< charT > run() ;
  basic_command_line_parser & allow_unregistered() ;
  basic_command_line_parser & extra_style_parser(style_parser) ;
};
```

Description

Command line parser.

The class allows one to specify all the information needed for parsing and to parse the command line. It is primarily needed to emulate named function parameters -- a regular function with 5 parameters will be hard to use and creating overloads with a smaller nuber of parameters will be confusing.

For the most common case, the function parse_command_line is a better alternative.

There are two typedefs -- command_line_parser and wcommand_line_parser, for charT == char and charT == wchar_t cases.

`basic_command_line_parser` construct/copy/destruct

1. ```
   basic_command_line_parser(const std::vector< std::basic_string< charT > > & arg
   ```

   Creates a command line parser for the specified arguments list. The 'args' parameter should not include program name.

2. ```
   basic_command_line_parser(int argc, charT * argv);
   ```

   Creates a command line parser for the specified arguments list. The parameters should be the same as passed to 'main'.

`basic_command_line_parser` public member functions

1. ```
   basic_command_line_parser & options(const options_description & desc) ;
   ```
   ```

Sets options descriptions to use.

2.
```
basic_command_line_parser &
positional(const positional_options_description & desc) ;
```

Sets positional options description to use.

3.
```
basic_command_line_parser & style(int ) ;
```

Sets the command line style.

4.
```
basic_command_line_parser & extra_parser(ext_parser ) ;
```

Sets the extra parsers.

5.
```
basic_parsed_options< charT > run() ;
```

Parses the options and returns the result of parsing. Throws on error.

6.
```
basic_command_line_parser & allow_unregistered() ;
```

Specifies that unregistered options are allowed and should be passed though. For each command like token that looks like an option but does not contain a recognized name, an instance of basic_option<charT> will be added to result, with 'unrecognized' field set to 'true'. It's possible to collect all unrecognized options with the 'collect_unrecognized' funciton.

7.
```
basic_command_line_parser & extra_style_parser(style_parser s) ;
```

Type collect_unrecognized_mode

Type collect_unrecognized_mode --

```
enum collect_unrecognized_mode { include_positional, exclude_positional };
```

Function template parse_command_line

Function template parse_command_line --

```
template<typename charT>
  basic_parsed_options< charT >
  parse_command_line(int argc, charT * argv, const options_description & ,
                     int style = 0,
                     function1< std::pair< std::string, std::string >, const std::
```

Description

Creates instance of 'command_line_parser', passes parameters to it, and returns the result of calling the 'run' method.

Function template parse_config_file

Function template parse_config_file --

```
template<typename charT>
  BOOST_PROGRAM_OPTIONS_DECL basic_parsed_options< charT >
  parse_config_file(std::basic_istream< charT > & ,
                    const options_description & );
```

Description

Parse a config file.

Function template collect_unrecognized

Function template collect_unrecognized --

```
template<typename charT>
  std::vector< std::basic_string< charT > >
  collect_unrecognized(const std::vector< basic_option< charT > > & options,
                       enum collect_unrecognized_mode mode);
```

Description

Collects the original tokens for all named options with 'unregistered' flag set. If 'mode' is 'include_positional' also collects all positional options. Returns the vector of origianl tokens for all collected options.

Function parse_environment

Function parse_environment --

```
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & ,
                  const function1< std::string, std::string > & name_mapper);
```

Description

Parse environment.

For each environment variable, the 'name_mapper' function is called to obtain the option name. If it returns empty string, the variable is ignored.

This is done since naming of environment variables is typically different from the naming of command line options.

Function parse_environment

Function parse_environment --

```
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & , const std::string & prefix);
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & , const char * prefix);
```

Description

Parse environment.

Takes all environment variables which start with 'prefix'. The option name is obtained from variable name by removing the prefix and converting the remaining string into lower case.

Header <boost/program_options/positional_options.hpp>

```
namespace boost {
  namespace program_options {
    class positional_options_description;
  }
}
```

Class positional_options_description

Class positional_options_description --

```
class positional_options_description {
public:
  // construct/copy/destruct
  positional_options_description();

  // public member functions
  void add(const char *, int) ;
  unsigned max_total_count() const;
  const std::string & name_for_position(unsigned) const;
};
```

Description

Describes positional options.

The class allows to guess option names for positional options, which are specified on the command line and are identified by the position. The class uses the information provided by the user to associate a name with every positional option, or tell that no name is known.

The primary assumption is that only the relative order of the positional options themselves matters, and that any interleaving ordinary options don't affect interpretation of positional options.

The user initializes the class by specifying that first N positional options should be given the name X1, following M options should be given the name X2 and so on.

`positional_options_description` construct/copy/destruct

1.
```
positional_options_description();
```

`positional_options_description` public member functions

1.
```
void add(const char * name, int max_count) ;
```

Species that up to 'max_count' next positional options should be given the 'name'. The value of '-1' means 'unlimited'. No calls to 'add' can be made after call with 'max_value' equal to '-1'.

2.
```
unsigned max_total_count() const;
```

Returns the maximum number of positional options that can be present. Can return numeric_limits<unsigned>::max() to indicate unlimited number.

3.
```
const std::string & name_for_position(unsigned position) const;
```

Returns the name that should be associated with positional options at 'position'. Precondition: position < max_total_count()

Header <boost/program_options/value_semantic.hpp>

```
namespace boost {
  namespace program_options {
    class value_semantic;
    template<typename charT> class value_semantic_codecvt_helper;

    template<> class value_semantic_codecvt_helper<char>;
    template<> class value_semantic_codecvt_helper<wchar_t>;

    class untyped_value;
    template<typename T, typename charT = char> class typed_value;
    template<typename T> typed_value< T > * value();
    template<typename T> typed_value< T > * value(T *);
    template<typename T> typed_value< T, wchar_t > * wvalue();
    template<typename T> typed_value< T, wchar_t > * wvalue(T *);
    BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch();
    BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch(bool *);
  }
}
```

Class value_semantic

Class value_semantic --

```
class value_semantic {
public:
  // construct/copy/destruct
  ~value_semantic();

  // public member functions
  virtual std::string name() const;
  virtual unsigned min_tokens() const;
  virtual unsigned max_tokens() const;
  virtual bool is_composing() const;
  virtual void
  parse(boost::any &, const std::vector< std::string > &, bool) const;
  virtual bool apply_default(boost::any &) const;
  virtual void notify(const boost::any &) const;
};
```

Description

Class which specifies how the option's value is to be parsed and converted into C++ types.

`value_semantic` construct/copy/destruct

1. ```
   ~value_semantic();
   ```

`value_semantic` public member functions

1. ```
   virtual std::string name() const;
   ```

   Returns the name of the option. The name is only meaningful for automatic help message.

2. ```
   virtual unsigned min_tokens() const;
   ```

   The minimum number of tokens for this option that should be present on the command line.

3. ```
   virtual unsigned max_tokens() const;
   ```

   The maximum number of tokens for this option that should be present on the command line.

4. ```
   virtual bool is_composing() const;
   ```

   Returns true if values from different sources should be composed. Otherwise, value from the first
   ```

source is used and values from other sources are discarded.

5.
```
virtual void
parse(boost::any & value_store, const std::vector< std::string > & new_tokens,
      bool utf8) const;
```

Parses a group of tokens that specify a value of option. Stores the result in 'value_store', using whatever representation is desired. May be be called several times if value of the same option is specified more than once.

6.
```
virtual bool apply_default(boost::any & value_store) const;
```

Called to assign default value to 'value_store'. Returns true if default value is assigned, and false if no default value exists.

7.
```
virtual void notify(const boost::any & value_store) const;
```

Called when final value of an option is determined.

Class template value_semantic_codecvt_helper

Class template value_semantic_codecvt_helper --

```
template<typename charT>
class value_semantic_codecvt_helper {
public:
};
```

Description

Helper class which perform necessary character conversions in the 'parse' method and forwards the data further.

Specializations

- Class value_semantic_codecvt_helper<char>
- Class value_semantic_codecvt_helper<wchar_t>

Class value_semantic_codecvt_helper<char>

Class value_semantic_codecvt_helper<char> --

```
class value_semantic_codecvt_helper<char> {
public:

  // protected member functions
  virtual void xparse(boost::any &, const std::vector< std::string > &) const;

  // private member functions
  void parse(boost::any &, const std::vector< std::string > &, bool) const;
};
```

Description

value_semantic_codecvt_helper protected member functions

1.
```
virtual void
xparse(boost::any & value_store,
       const std::vector< std::string > & new_tokens) const;
```

value_semantic_codecvt_helper private member functions

1.
```
void parse(boost::any & value_store,
           const std::vector< std::string > & new_tokens, bool utf8) const;
```

Class value_semantic_codecvt_helper<wchar_t>

Class value_semantic_codecvt_helper<wchar_t> --

```
class value_semantic_codecvt_helper<wchar_t>
  :  : public boost::program_options::value_semantic
{
public:

  // protected member functions
  virtual void xparse(boost::any &, const std::vector< std::wstring > &) const;

  // private member functions
  void parse(boost::any &, const std::vector< std::string > &, bool) const;
};
```

Description

value_semantic_codecvt_helper protected member functions

1.
```
virtual void
xparse(boost::any & value_store,
       const std::vector< std::wstring > & new_tokens) const;
```

value_semantic_codecvt_helper private member functions

1.
```
void parse(boost::any & value_store,
           const std::vector< std::string > & new_tokens, bool utf8) const;
```

Class untyped_value

Class untyped_value --

```
class untyped_value
  :  : public boost::program_options::value_semantic_codecvt_helper< charT >
{
public:
  // construct/copy/destruct
  untyped_value(bool = false);

  // public member functions
  std::string name() const;
  unsigned min_tokens() const;
  unsigned max_tokens() const;
  bool is_composing() const;
  void xparse(boost::any &, const std::vector< std::string > &) const;
  bool apply_default(boost::any &) const;
  void notify(const boost::any &) const;
};
```

Description

Class which specifies a simple handling of a value: the value will have string type and only one token is allowed.

`untyped_value` construct/copy/destruct

1. ```
   untyped_value(bool zero_tokens = false);
   ```

`untyped_value` public member functions

1. ```
   std::string name() const;
   ```

2. ```
   unsigned min_tokens() const;
   ```

3. ```
   unsigned max_tokens() const;
   ```

4. ```
   bool is_composing() const;
   ```

5. ```
   void xparse(boost::any & value_store,
               const std::vector< std::string > & new_tokens) const;
   ```

If 'value_store' is already initialized, or new_tokens has more than one elements, throws. Otherwise, assigns the first string from 'new_tokens' to 'value_store', without any modifications.

6.
```
bool apply_default(boost::any & ) const;
```

Does nothing.

7.
```
void notify(const boost::any & ) const;
```

Does nothing.

Class template typed_value

Class template typed_value --

```
template<typename T, typename charT = char>
class typed_value
  :  : public boost::program_options::value_semantic_codecvt_helper< charT >
{
public:
  // construct/copy/destruct
  typed_value(T *);

  // public member functions
  typed_value * default_value(const T &) ;
  typed_value * default_value(const T &, const std::string &) ;
  typed_value * notifier(function1< void, const T & >) ;
  typed_value * composing() ;
  typed_value * multitoken() ;
  typed_value * zero_tokens() ;
  std::string name() const;
  bool is_composing() const;
  unsigned min_tokens() const;
  unsigned max_tokens() const;
  void xparse(boost::any &, const std::vector< std::basic_string< charT > > &) con
  virtual bool apply_default(boost::any &) const;
  void notify(const boost::any &) const;
};
```

Description

Class which handles value of a specific type.

`typed_value` construct/copy/destruct

1.
```
typed_value(T * store_to);
```

Ctor. The 'store_to' parameter tells where to store the value when it's known. The parameter can be NULL.

`typed_value` public member functions

1.
```
typed_value * default_value(const T & v) ;
```

Specifies default value, which will be used if none is explicitly specified. The type 'T' should provide operator<< for ostream.

2.
```
typed_value * default_value(const T & v, const std::string & textual) ;
```

Specifies default value, which will be used if none is explicitly specified. Unlike the above overload, the type 'T' need not provide operator<< for ostream, but textual representation of default

value must be provided by the user.

3. `typed_value * notifier(function1< void, const T & > f) ;`

   Specifies a function to be called when the final value is determined.

4. `typed_value * composing() ;`

   Specifies that the value is composing. See the 'is_composing' method for explanation.

5. `typed_value * multitoken() ;`

   Specifies that the value can span multiple tokens.

6. `typed_value * zero_tokens() ;`

7. `std::string name() const;`

8. `bool is_composing() const;`

9. `unsigned min_tokens() const;`

10. `unsigned max_tokens() const;`

11. ```
    void xparse(boost::any & value_store,
                const std::vector< std::basic_string< charT > > & new_tokens) const
    ```

    Creates an instance of the 'validator' class and calls its operator() to perform athe ctual conversion.

12. `virtual bool apply_default(boost::any & value_store) const;`

    If default value was specified via previous call to 'default_value', stores that value into 'value_store'. Returns true if default value was stored.

13. `void notify(const boost::any & value_store) const;`

    If an address of variable to store value was specified when creating *this, stores the value there. Otherwise, does nothing.
    ```

Function value

Function value --

```
template<typename T> typed_value< T > * value();
template<typename T> typed_value< T > * value(T * v);
```

Description

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor. The second overload is used when it's additionally desired to store the value of option into program variable.

Function wvalue

Function wvalue --

```
template<typename T> typed_value< T, wchar_t > * wvalue();
template<typename T> typed_value< T, wchar_t > * wvalue(T * v);
```

Description

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor.

Function bool_switch

Function bool_switch --

```
BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch();
BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch(bool * v);
```

Description

Works the same way as the 'value<bool>' function, but the created value_semantic won't accept any explicit value. So, if the option is present on the command line, the value will be 'true'.

Header <boost/program_options/variables_map.hpp>

```
namespace boost {
  namespace program_options {
    class variable_value;
    class abstract_variables_map;
    class variables_map;
    BOOST_PROGRAM_OPTIONS_DECL void
    store(const basic_parsed_options< char > &, variables_map &, bool = false);
    BOOST_PROGRAM_OPTIONS_DECL void
    store(const basic_parsed_options< wchar_t > &, variables_map &);
    BOOST_PROGRAM_OPTIONS_DECL void notify(variables_map &);
  }
}
```

Class variable_value

Class variable_value --

```
class variable_value {
public:
  // construct/copy/destruct
  variable_value();
  variable_value(const boost::any &, bool);

  // public member functions
  template<typename T> const T & as() const;
  template<typename T> T & as() ;
  bool empty() const;
  bool defaulted() const;
  const boost::any & value() const;
  boost::any & value() ;
};
```

Description

Class holding value of option. Contains details about how the value is set and allows to conveniently obtain the value.

`variable_value` construct/copy/destruct

1. `variable_value();`

2. `variable_value(const boost::any & v, bool defaulted);`

`variable_value` public member functions

1. `template<typename T> const T & as() const;`

   If stored value if of type T, returns that value. Otherwise, throws boost::bad_any_cast exception.

2. `template<typename T> T & as() ;`

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3. `bool empty() const;`

4. `bool defaulted() const;`

Returns true if the value was not explicitly given, but has default value.

5.
```
const boost::any & value() const;
```

Returns the contained value.

6.
```
boost::any & value() ;
```

Returns the contained value.

Class abstract_variables_map

Class abstract_variables_map --

```
class abstract_variables_map {
public:
  // construct/copy/destruct
  abstract_variables_map();
  abstract_variables_map(const abstract_variables_map *);
  ~abstract_variables_map();

  // public member functions
  const variable_value & operator[](const std::string &) const;
  void next(abstract_variables_map *) ;

  // private member functions
  virtual const variable_value & get(const std::string &) const;
};
```

Description

Implements string->string mapping with convenient value casting facilities.

`abstract_variables_map` construct/copy/destruct

1. ```
   abstract_variables_map();
   ```

2. ```
   abstract_variables_map(const abstract_variables_map * next);
   ```

3. ```
   ~abstract_variables_map();
   ```

`abstract_variables_map` public member functions

1. ```
   const variable_value & operator[](const std::string & name) const;
   ```

   Obtains the value of variable 'name', from *this and possibly from the chain of variable maps.

   - if there's no value in *this.
     - if there's next variable map, returns value from it
     - otherwise, returns empty value
   - if there's defaulted value

- if there's next varaible map, which has a non-defauled value, return that
- otherwise, return value from *this

- if there's a non-defauled value, returns it.

2. ```
   void next(abstract_variables_map * next) ;
   ```

   Sets next variable map, which will be used to find variables not found in *this.

`abstract_variables_map` private member functions

1. ```
   virtual const variable_value & get(const std::string & name) const;
   ```

   Returns value of variable 'name' stored in *this, or empty value otherwise.

Class variables_map

Class variables_map --

```
class variables_map
  :  : public boost::program_options::abstract_variables_map
{
public:
  // construct/copy/destruct
  variables_map();
  variables_map(const abstract_variables_map *);

  // public member functions
  const variable_value & operator[](const std::string &) const;

  // private member functions
  const variable_value & get(const std::string &) const;
};
```

Description

Concrete variables map which store variables in real map.

`variables_map` construct/copy/destruct

1. ```
   variables_map();
   ```

2. ```
   variables_map(const abstract_variables_map * next);
   ```

`variables_map` public member functions

1. ```
   const variable_value & operator[](const std::string & name) const;
   ```

   Obtains the value of variable 'name', from *this and possibly from the chain of variable maps.

   - if there's no value in *this.
     - if there's next variable map, returns value from it
     - otherwise, returns empty value
   - if there's defaulted value
     - if there's next varaible map, which has a non-defauled value, return that
     - otherwise, return value from *this
   ```

- if there's a non-defauled value, returns it.

`variables_map` private member functions

1. ```
   const variable_value & get(const std::string & name) const;
   ```

   Implementation of abstract_variables_map::get which does 'find' in *this.
   ```

Function store

Function store --

```
BOOST_PROGRAM_OPTIONS_DECL void
store(const basic_parsed_options< char > & options, variables_map & m,
      bool utf8 = false);
```

Description

Stores in 'm' all options that are defined in 'options'. If 'm' already has a non-defaulted value of an option, that value is not changed, even if 'options' specify some value.

Function store

Function store --

```
BOOST_PROGRAM_OPTIONS_DECL void
store(const basic_parsed_options< wchar_t > & options, variables_map & m);
```

Description

Stores in 'm' all options that are defined in 'options'. If 'm' already has a non-defaulted value of an option, that value is not changed, even if 'options' specify some value. This is wide character variant.

Function notify

Function notify --

```
BOOST_PROGRAM_OPTIONS_DECL void notify(variables_map & m);
```

Description

Runs all 'notify' function for options in 'm'.

Header <boost/program_options/version.hpp>

```
BOOST_PROGRAM_OPTIONS_VERSION
```

Macro BOOST_PROGRAM_OPTIONS_VERSION

Macro BOOST_PROGRAM_OPTIONS_VERSION --

```
BOOST_PROGRAM_OPTIONS_VERSION
```

Description

The version of the source interface. The value will be incremented whenever a change is made which might cause compilation errors for existing code.