Chapter 63. Boost.ProgramOptions

Boost.ProgramOptions is a library that makes it easy to parse command-line options, for example, for console applications. If you develop applications with a graphical user interface, command-line options are usually not important.

To parse command-line options with Boost.ProgramOptions, the following three steps are required:

1. Define command-line options. You give them names and specify which ones can be set to a value. If a command-line option is parsed as a key/value pair, you also set the type of the value – for example, whether it is a string or a number.

2. Use a parser to evaluate the command line. You get the command line from the two parameters of `main()`, which are usually called **`argc`** and **`argv`**.

3. Store the command-line options evaluated by the parser. Boost.ProgramOptions offers a class derived from `std::map` that saves command-line options as name/value pairs. Afterwards, you can check which options have been stored and what their values are.

Example 63.1 shows the basic approach for parsing command-line options with Boost.ProgramOptions.

Example 63.1. Basic approach with Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <iostream>

using namespace boost::program_options;

void on_age(int age)
{
  std::cout << "On age: " << age << '\n';
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->default_value(3.14f), "Pi")
      ("age", value<int>()->notifier(on_age), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << vm["age"].as<int>() << '\n';
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
```

```
    std::cerr << ex.what() << '\n';
  }
}
```

To use Boost.ProgramOptions, include the header file `boost/program_options.hpp`. You can access all classes and functions from this library in the namespace `boost::program_options`.

Use the class `boost::program_options::options_description` to describe command-line options. An object of this type can be written to a stream such as **`std::cout`** to display an overview of available command-line options. The string passed to the constructor gives the overview a name that acts as a title for the command-line options.

`boost::program_options::options_description` defines a member function `add()` that expects a parameter of type `boost::program_options::option_description`. You call this function to describe each command-line option. Instead of calling this function for every command-line option, Example 63.1 calls the member function `add_options()`, which makes that task easier.

`add_options()` returns a proxy object representing an object of type `boost::program_options::options_description`. The type of the proxy object doesn't matter. It's more interesting that the proxy object simplifies defining many command-line options. It uses the overloaded operator `operator()`, which you can call to pass the required data to define a command-line option. This operator returns a reference to the same proxy object, which allows you to call `operator()` multiple times.

Example 63.1 defines three command-line options with the help of the proxy object. The first command-line option is `--help`. The description of this option is set to "Help screen". The option is a switch, not a name/value pair. You set `--help` on the command line or omit it. It's not possible to set `--help` to a value.

Please note that the first string passed to `operator()` is "help,h". You can specify short names for command-line options. A short name must consist of just one letter and is set after a comma. Now the help can be displayed with either `--help` or `-h`.

Besides `--help`, two more command-line options are defined: `--pi` and `--age`. These options aren't switches, they're name/value pairs. Both `--pi` and `--age` expect to be set to a value.

You pass a pointer to an object of type `boost::program_options::value_semantic` as the second parameter to `operator()` to define an option as a name/value pair. You don't need to access `boost::program_options::value_semantic` directly. You can use the helper function `boost::program_options::value()`, which creates an object of type `boost::program_options::value_semantic`. `boost::program_options::value()` returns the object's address, which you then can pass to the proxy object using `operator()`.

`boost::program_options::value()` is a function template that takes the type of the command-line option value as a template parameter. Thus, the command-line option `--age` expects an integer and `--pi` expects a floating point number.

The object returned from `boost::program_options::value()` provides some useful member functions. For example, you can call `default_value()` to provide a default value. Example 63.1 sets `--pi` to 3.14 if that option isn't used on the command line.

`notifier()` links a function to a command-line option's value. That function is then called with the value of the command-line option. In Example 63.1, the function `on_age()` is linked to `--age`. If the command-line option `--age` is used to set an age, the age is passed to `on_age()` which writes it to standard output.

Processing values with functions like `on_age()` is optional. You don't have to use `notifier()` because it's possible to access values in other ways.

After all command-line options have been defined, you use a parser. In Example 63.1, the helper function `boost::program_options::parse_command_line()` is called to parse the command line. This function takes **`argc`** and **`argv`**, which define the command line, and **`desc`**, which contains the option descriptions. `boost::program_options::parse_command_line()` returns the parsed options in an object of type `boost::program_options::parsed_options`. You usually don't access this object directly. Instead you pass it to `boost::program_options::store()`, which stores the parsed options in a container.

Example 63.1 passes **`vm`** as a second parameter to `boost::program_options::store()`. **`vm`** is an object of type `boost::program_options::variables_map`. This class is derived from the class `std::map<std::string, boost::program_options::variable_value>` and, thus, provides the same member functions as `std::map`. For example, you can call `count()` to check whether a certain command-line option has been used and is stored in the container.

In Example 63.1, before **`vm`** is accessed and `count()` is called, `boost::program_options::notify()` is called. This function triggers functions, such as `on_age()`, that are linked to a value using `notifier()`. Without `boost::program_options::notify()`, `on_age()` would not be called.

`vm` lets you check whether a certain command-line option exists, and it also lets you access the value the command-line option is set to. The value's type is `boost::program_options::variable_value`, a class that uses `boost::any` internally. You can get the object of type `boost::any` from the member function `value()`.

Example 63.1 calls `as()`, not `value()`. This member function converts the value of a command-line option to the type passed as a template parameter. `as()` uses `boost::any_cast()` for the type conversion.

Be sure the type you pass to `as()` matches the type of the command-line option. For example, Example 63.1 expects the command-line option `--age` to be set to a number of type `int`, so `int` must be passed as a template parameter to `as()`.

You can start Example 63.1 in many ways. Here is one example:

test

In this case `Pi: 3.14` is displayed. Because `--pi` isn't set on the command line, the default value is displayed.

This example sets a value using `--pi`:

test --pi 3.1415

The program now displays `Pi: 3.1415`.

This example also passes an age:

test --pi 3.1415 --age 29

The output is now `On age: 29` and `Age: 29`. The first line is written when `boost::program_options::notify()` is called; this triggers the execution of `on_age()`. There is no output for `--pi` because the program uses `else if` statements that only display the value set with `--pi` if `--age` is not set.

This example shows the help:

test -h

You get a complete overview on all command-line options:

```
Options: -h [ --help ]          Help screen --pi arg (=3.1400001) Pi --age arg
```

As you can see, the help can be shown in two different ways because a short name for that command-line option was defined. For `--pi` the default value is displayed. The command-line options and their descriptions are formatted automatically. You only need to write the object of type `boost::program_options::options_description` to standard output as in Example 63.1.

Now, start the example like this:

test --age

The output is the following:

```
the required argument for option '--age' is missing.
```

Because `--age` isn't set, the parser used in `boost::program_options::parse_command_line()` throws an exception of type `boost::program_options::error`. The exception is caught, and an error message is written to standard output.

`boost::program_options::error` is derived from `std::logic_error`. Boost.ProgramOptions defines additional exceptions, which are all derived from `boost::program_options::error`. One of those exceptions is `boost::program_options::invalid_syntax`, which is the exact exception thrown in Example 63.1 if you don't supply a value for `--age`.

Example 63.2 introduces more configuration settings available with Boost.ProgramOptions.

Example 63.2. Special configuration settings with Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(), std::ostream_iterator<std::string>{
    std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    int age;

    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->implicit_value(3.14f), "Pi")
      ("age", value<int>(&age), "Age")
      ("phone", value<std::vector<std::string>>()->multitoken()->
        zero_tokens()->composing(), "Phone")
      ("unreg", "Unrecognized options");

    command_line_parser parser{argc, argv};
    parser.options(desc).allow_unregistered().style(
      command_line_style::default_style |
      command_line_style::allow_slash_for_short);
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << age << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
    else if (vm.count("unreg"))
      to_cout(collect_unrecognized(parsed_options.options,
        exclude_positional));
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.2 parses command-line options like the previous example does. However, there are some notable differences. For example, `implicit_value()` is called, rather than `default_value()`, when defining the `--pi` command-line option. This means that pi isn't set to

3.14 by default. `--pi` must be set on the command line for pi to be available. However, you don't need to supply a value to the `--pi` command-line option if you use `implicit_value()`. It's sufficient to pass `--pi` without setting a value. In that case, pi is set to 3.14 implicitly.

For the command-line option `--age`, a pointer to the variable **age** is passed to `boost::program_options::value()`. This stores the value of a command-line option in a variable. Of course, the value is still available in the container **vm**.

Please note that a value is only stored in **age** if `boost::program_options::notify()` is called. Even though `notifier()` isn't used in this example, `boost::program_options::notify()` still must be used. To avoid problems, it's a good idea to always call `boost::program_options::notify()` after parsed command-line options have been stored with `boost::program_options::store()`.

Example 63.2 supports a new command-line option `--phone` to pass a phone number to the program. In fact, you can pass multiple phone numbers on the command line. For example, the following command line starts the program with the phone numbers 123 and 456:

test --phone 123 456

Example 63.2 supports multiple phone numbers because `multitoken()` is called on this command-line option's value. And, since `zero_tokens()` is called, `--phone` can also be used without passing a phone number.

You can also pass multiple phone numbers by repeating the `--phone` option, as shown in the following command line:

test --phone 123 --phone 456

In this case, both phone numbers, 123 and 456, are parsed. The call to `composing()` makes it possible to use a command-line option multiple times – the values are composed.

The value of the argument to `--phone` is of type `std::vector<std::string>`. You need to use a container to store multiple phone numbers.

Example 63.2 defines another command-line option, `--unreg`. This is a switch that can't be set to a value. It is used later in the example to decide whether command-line options that aren't defined in **desc** should be displayed.

While Example 63.1 calls the function `boost::program_options::parse_command_line()` to parse command-line options, Example 63.2 uses a parser of type `boost::program_options::command_line_parser`. **argc** and **argv** are passed to the constructor.

`boost::program_options::command_line_parser` provides several member functions. You must call `options()` to pass the definition of command-line options to the parser.

Like other member functions, `options()` returns a reference to the same parser. That way, member functions can be easily called one after another. Example 63.2 calls

`allow_unregistered()` after `options()` to tell the parser not to throw an exception if unknown command-line options are detected. Finally, `style()` is called to tell the parser that short names can be used with a slash. Thus, the short name for the `--help` option can be either `-h` or `/h`.

Please note that `boost::program_options::parse_command_line()` supports a fourth parameter, which is forwarded to `style()`. If you want to use an option like **`boost::program_options::command_line_style::allow_slash_for_short`**, you can still use the function `boost::program_options::parse_command_line()`.

After the configuration has been set, call `run()` on the parser. This member function returns the parsed command-line options in an object of type `boost::program_options::parsed_options`, which you can pass to `boost::program_options::store()` to store the options in **`vm`**.

Later in the code, Example 63.2 accesses **`vm`** again to evaluate command-line options. Only the call to `boost::program_options::collect_unrecognized()` is new. This function is called for the command-line option `--unreg`. The function expects an object of type `boost::program_options::parsed_options`, which is returned by `run()`. It returns all unknown command-line options in a `std::vector<std::string>`. For example, if you start the program with **test --unreg --abc**, `--abc` will be written to standard output.

When **`boost::program_options::exclude_positional`** is passed as the second parameter to `boost::program_options::collect_unrecognized()`, positional options are ignored. For Example 63.2, this doesn't matter because no positional options are defined. However, `boost::program_options::collect_unrecognized()` requires this parameter.

Example 63.3 illustrates positional options.

Example 63.3. Positional options with Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(),
    std::ostream_iterator<std::string>{std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("phone", value<std::vector<std::string>>()->
        multitoken()->zero_tokens()->composing(), "Phone");

    positional_options_description pos_desc;
    pos_desc.add("phone", -1);
```

```
    command_line_parser parser{argc, argv};
    parser.options(desc).positional(pos_desc).allow_unregistered();
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.3 defines `--phone` as a positional option using the class `boost::program_options::positional_options_description`. This class provides the member function `add()`, which expects the name of the command-line option and a position to be passed. The example passes “phone” and -1.

With positional options, values can be set on the command line without using command-line options. You can start Example 63.3 like this:

test 123 456

Even though `--phone` isn’t used, 123 and 456 are recognized as phone numbers.

Calling `add()` on an object of type `boost::program_options::positional_options_description` assigns values on the command line to command-line options using position numbers. When Example 63.3 is called using the command line **test 123 456**, 123 has the position number 0 and 456 has the position number 1. Example 63.3 passes -1 to `add()`, which assigns all of the values – 123 and 456 – to `--phone`. If you changed Example 63.3 to pass the value 0 to `add()`, only 123 would be recognized as a phone number. And if 1 was passed to `add()`, only 456 would be recognized.

`pos_desc` is passed with `positional()` to the parser. That’s how the parser knows which command-line options are positional.

Please note that you have to make sure that positional options are defined. In Example 63.3, for example, “phone” could only be passed to `add()` because a definition for `--phone` already existed in **`desc`**.

In all previous examples, Boost.ProgramOptions was used to parse command-line options. However, the library supports loading configuration options from a file, too. This can be useful if the same command-line options have to be set repeatedly.

Example 63.4. Loading options from a configuration file

```
#include <boost/program_options.hpp>
#include <string>
```

```
#include <fstream>
#include <iostream>

using namespace boost::program_options;

int main(int argc, const char *argv[])
{
  try
  {
    options_description generalOptions{"General"};
    generalOptions.add_options()
      ("help,h", "Help screen")
      ("config", value<std::string>(), "Config file");

    options_description fileOptions{"File"};
    fileOptions.add_options()
      ("age", value<int>(), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, generalOptions), vm);
    if (vm.count("config"))
    {
      std::ifstream ifs{vm["config"].as<std::string>().c_str()};
      if (ifs)
        store(parse_config_file(ifs, fileOptions), vm);
    }
    notify(vm);

    if (vm.count("help"))
      std::cout << generalOptions << '\n';
    else if (vm.count("age"))
      std::cout << "Your age is: " << vm["age"].as<int>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.4 uses two objects of type `boost::program_options::options_description`. **`generalOptions`** defines options that must be set on the command line. **`fileOptions`** defines options that can be loaded from a configuration file.

It's not mandatory to define options with two different objects of type `boost::program_options::options_description`. You can use just one if the set of options is the same for both command line and file. In Example 63.4, separating options makes sense because you don't want to allow `--help` to be set in the configuration file. If that was allowed and the user put that option in the configuration file, the program would display the help screen every time.

Example 63.4 loads `--age` from a configuration file. You can pass the name of the configuration file as a command-line option. In this example, `--config` is defined in **`generalOptions`** for that reason.

After the command-line options have been parsed with `boost::program_options::parse_command_line()` and stored in **`vm`**, the example checks whether `--config` is set. If it is, the configuration file is opened with `std::ifstream`. The `std::ifstream` object is passed to the function `boost::program_options::parse_config_file()` along with **`fileOptions`**, which describes

the options. `boost::program_options::parse_config_file()` does the same thing as `boost::program_options::parse_command_line()` and returns parsed options in an object of type `boost::program_options::parsed_options`. This object is passed to `boost::program_options::store()` to store the parsed options in **`vm`**.

If you create a file called `config.txt`, put age=29 in that file, and execute the command line below, you will get the result shown.

test --config config.txt

The output is the following:

```
Your age is: 29
```

If you support the same options on the command line and in a configuration file, your program may parse the same option twice – once with `boost::program_options::parse_command_line()` and once with `boost::program_options::parse_config_file()`. The order of the function calls determines which value you will find in **`vm`**. Once a command-line option's value has been stored in **`vm`**, that value will not be overwritten. Whether the value is set by an option on the command line or in a configuration file depends only on the order in which you call the `store()` function.

Boost.ProgramOptions also defines the function `boost::program_options::parse_environment()`, which can be used to load options from environment variables. The class `boost::environment_iterator` lets you iterate over environment variables.