tangentsky Documentation

Release 0.1.0

Josh Bialkowski

Mar 20, 2020

Contents

`argue` is an argument parsing library for C++, largely inspired by python's `argparse` package.

CHAPTER 1

Features

1.1 Automatic Help

`argue` parsers can automatically pretty-print help text, such as this:

```
==========
argue-demo
==========
version: 0.1.3-dev5
author : Josh Bialkowski <josh.bialkowski@gmail.com>
copyright: (C) 2018

argue-demo [-h/--help] [-v/--version] [-s/--sum] <N> [N..]

Flags:
------
-h  --help          print this help message
-v  --version       print version information and exit
-s  --sum           sum the integers (default: find the max)

Positionals:
------------
integer             an integer for the accumulator
                    choices=[1, 2, 3, 4]
```

This action is automatically added to the parser with the flags `-h/--help` if the `add_help` metadata option is `true`.

1.2 Machine Parseable Help

The help information can be printed in JSON format instead of pretty text. The JSON can then be processed to generate documentation pages (man pages or HTML). Just export `ARGUE_HELP_FORMAT="json"` in the environment

before using `--help`. If the program uses subparsers, you may also wish to export `ARGUE_HELP_RECURSE="1"` to include help contents for all subparsers recursively.

The JSON help for the demo program is:

```
[
{
  "metadata": {
    "id": "0x7ffc11b0cb40",
    "name": "argue-demo",
    "author": "Josh Bialkowski <josh.bialkowski@gmail.com>",
    "copyright": "(C) 2018",
    "prolog": "",
    "epilog": "",
    "comamnd_prefix": "",
    "subdepth": 0,
    "usage": "argue-demo [-h/--help] [-v/--version] [-s/--sum] <N> [N..]\n"
  },
  "flags": [
    {
      "short_flag": "-h",
      "long_flag": "--help",
      "help": "print this help message"
    },
    {
      "short_flag": "-v",
      "long_flag": "--version",
      "help": "print version information and exit"
    },
    {
      "short_flag": "-s",
      "long_flag": "--sum",
      "help": "sum the integers (default: find the max)"
    }
  ],
  "positional": [
    {
      "name": "integer",
      "help": "an integer for the accumulator\nchoices=[1, 2, 3, 4]"
    }
  ],
  "subcommands": [
]
}]
```

1.3 Subcommands / Subparsers

Argue supports arbitrary nesting of subcommands via `ArgumentParser::add_subparsers`. The API mirrors that of python's `argparse`. See `examples/subparser_example.cc` for an example. The help text for this example is:

```
=================
subparser-example
=================

subparser-example [-h/--help] <command>
```

(continues on next page)

(continued from previous page)

```
Flags:
------
-h  --help          print this help message

Positionals:
------------
command             [bar, foo]

Subcommands:
------------
bar                 The bar command does bar
foo                 The foo command does foo
```

1.4 Automatic Bash Completion

Any program which uses `argue` to parse it's command line arguments will automatically supports bash auto-completion. The completion script can be found at `bash_completion.d/argue-autocomplete`. Install this anywhere that bash looks for completion scripts (e.g. `/usr/share/bash-completion` or `~/.bash_completion.d` if the user is so configured). The script only needs to be installed once, and completion will work for all `argue` enabled programs.

The completion script will detect if a program uses `argue` for it's argument parsing and, if it does, it will call the program with some additional environment variables signallying the `ArgumentParser` to work in completion mode instead of regular mode.

1.5 Unique Prefix Matching for Long Flags

If the program user provides a long flag (e.g. `--do-optional`) which does not match any known long flags, but is a unique prefix of a known flag (e.g. `--do-optional-thing`), then `argue` will match the known flag. The prefix must be unique so `--do` will not match if `--do-optional-thing` and `--do-other-thing` are both known.

CHAPTER 2

Usage

2.1 Creating the parser

To start with `argue`, create an `ArgumentParser` and use `add_argument` to populate arguments. Then use `parse_args` to do the parsing.

```
// Copyright 2020 Josh Bialkowski <josh.bialkowski@gmail.com>
#include "argue/argue.h"

int main(int argc, char** argv) {
  argue::Parser::Metadata meta{/*add_help=*/true};
  meta.name = "subparser-example";
  argue::Parser parser{meta};

  std::string foo;
  std::string bar;
  parser.add_argument("arg", &foo);
  parser.add_argument("-b", "--bar", &bar);

  int parse_result = parser.parse_args(argc, argv);
  switch (parse_result) {
    case argue::PARSE_ABORTED:
      return 0;
    case argue::PARSE_EXCEPTION:
      return 1;
    case argue::PARSE_FINISHED:
      break;
  }

  std::cout << "foo: " << foo << ", bar: " << bar << std::endl;
  return 0;
}
```

2.2 Specifying Argument Options

You can specify additional options to `add_argument` to configure how the parser should handle that argument.

There are three different APIs for adding arguments. The first two are experimental APIs designed to make your parser configuration code more compact and readable. They are, however, a little "magic", so there is a more intuitive fallback API.

2.2.1 Keyword API

They keyword API is meant to mimic the keyword arguments of the `argparse` python package. The `argue::keywords` namespace includes names of global objects which act as the "keywords". Use it like this:

```
std::string foo;
int bar;

// Import into the active namespace the names of the keyword objects, e.g.
// "action", "dest", "nargs', and "default_" used below.
using namespace argue::keywords;

// clang-format off
parser.add_argument(
  "-f", "--foo", action="store", dest=&foo, help="Foo does foo things");
parser.add_argument("-b", "--bar", dest=&bar, nargs="?", default_=1);
// clang-format on
```

Note: Because keyword arguments are not a common feature of C++ APIs, whatever beautifier you are using is likely to treat the keyword argument assigments as it would any other assignment. You may wish to locally disable your beautifier when using this API.

2.2.2 Kwargs Object API

If your compiler supports non-trivial designated initializers (e.g. clang 5+, or anything supporing c++20 designated initializers), then you can take advantage of this API. Additional options are provided by passing a `KWargs<T>` object after the destination argument. This object can be initialized inline resulting in a similar appearance to the above:

```
parser.add_argument("--foo", &foo, {.help="Foo does foo things"});
```

2.2.3 Fallback API

The `add_argument` overloads all return an argument object. You can directly assign the fields of this object to configure additional options.

```
auto arg = parser.add_argument("-f", "--foo", &foo);
arg.help = "Foo does foo things";
```

2.2.4 Additional Notes

You don't have to specify the destination inline as an argument after the name or flags. You could specify it as a keyword argument or as an assignment with the fallback API, however then the type of the argument cannot be

inferred during the call to `add_argument` and it must be provided explicitly. For example:

```
auto arg = parser.add_argument<int>("-f", "--foo");
arg.dest = &foo;
```

This is true for the KWargs object API and the fallback API, but is not required by the keywords API.

2.3 Supported Argument Options

Note: Much of the the text in this section is borrowed from https://docs.python.org/3/library/argparse.html

2.3.1 nargs

`.action=` may be either a `shared_ptr` to an `Action<T>` object, or it may be one of the following strings.

- `"store"` - This just stores the argument's value. This is the default action.
- `"store_const"` - This stores the value specified by the const keyword argument. The 'store_const' action is most commonly used with optional arguments that specify some sort of flag. For example:
- `"store_true"` and `"store_false"` - These are special cases of `"store_const"` used for storing the values True and False respectively. In addition, they create default values of False and True respectively.
- `"help"` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See ArgumentParser for details of how the output is created.
- `"version"` - This expects a `.version=` keyword argument in the `add_argument()` call, and prints version information and exits when invoked

2.3.2 nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The nargs keyword argument associates a different number of command-line arguments with a single action. The supported values are:

- `N` (an integer). `N` arguments from the command line will be gathered together into a list. Note that nargs=1 produces a list of one item. This is different from the default, in which the item is produced by itself.
- `"?"`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from const will be produced.
- `"*"`. All command-line arguments present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with nargs='*', but multiple optional arguments with nargs='*' is possible.
- `"+"`. Just like `"*"`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example:
- `argue::REMAINDER`. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities.

If the nargs keyword argument is not provided, the number of arguments consumed is determined by the action. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

Note that for `nargs` that imply a list of arguments, the destination object must be of a supported container type (e.g. `std::list` or `std::vector`).

2.3.3 const

The const argument of `add_argument()` is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions. The two most common uses of it are:

- When `add_argument()` is called with `.action="store_const"` or `.action="append_const"`. These actions add the const value to one of the attributes of the object returned by parse_args(). See the action description for examples.
- When `add_argument()` is called with option strings (like -f or –foo) and `nargs="?"`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of const will be assumed instead. See the nargs description for examples.

With the `"store_const"` and `"append_const"` actions, the const keyword argument must be given. For other actions, it defaults to None.

2.3.4 default_

All optional arguments and some positional arguments may be omitted at the command line. The default keyword argument of `add_argument()`, whose value defaults to None, specifies what value should be used if the command-line argument is not present. For optional arguments, the default value is used when the option string was not present at the command line.

Note that in C++ `default` is a reserved word so this keyword ends with an underscore ('_').

2.3.5 choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the choices keyword argument to `add_argument()`. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values

2.3.6 required

In general, `argue` assumes that flags like `-f` and `--bar` indicate optional arguments, which can always be omitted at the command line. To make an option required, `true` can be specified for the `required=` keyword argument to `add_argument()`.

2.3.7 help

The help value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these help descriptions will be displayed with each argument.

2.3.8 metavar

When `ArgumentParser` generates help messages, it needs some way to refer to each expected argument. By default, for arguments which have a flag, the flag name is used. Positional arguments have no default. In either case, a name can be specified with `metavar`.

2.3.9 dest

Most `ArgumentParser` actions store some value to some variable. The address of the variable to store values can be specifie dwith this keyword argument.

2.4 Demonstration

There are a couple of examples in the `examples/` directory of the source package. For example, here is a replica of the demo application from the python `argparse` documentation, written in C++ using `argue`:

```
// Copyright 2018 Josh Bialkowski <josh.bialkowski@gmail.com>
#include <iostream>
#include <list>
#include <memory>

#include "argue/argue.h"

class Accumulator {
 public:
  std::string GetName() {
    return name_;
  }
  virtual int operator()(const std::list<int>& args) = 0;

 protected:
  std::string name_;
};

struct Max : public Accumulator {
  Max() {
    name_ = "max";
  }

  int operator()(const std::list<int>& args) override {
    if (args.size() == 0) {
      return 0;
    }
    int result = args.front();
    for (int x : args) {
      if (x > result) {
        result = x;
      }
    }
    return result;
  }
};

struct Sum : public Accumulator {
```

(continues on next page)

(continued from previous page)

```
  Sum() {
    name_ = "sum";
  }

  int operator()(const std::list<int>& args) override {
    int result = 0;
    for (int x : args) {
      result += x;
    }
    return result;
  }
};

int main(int argc, char** argv) {
  std::list<int> int_args;
  std::shared_ptr<Accumulator> accumulate;
  std::shared_ptr<Accumulator> sum_fn = std::make_shared<Sum>();
  std::shared_ptr<Accumulator> max_fn = std::make_shared<Max>();

  argue::Parser parser({
      .add_help = true,
      .add_version = true,
      .name = "argue-demo",
      .version = argue::VersionString ARGUE_VERSION,
      .author = "Josh Bialkowski <josh.bialkowski@gmail.com>",
      .copyright = "(C) 2018",
  });

  using namespace argue::keywords;  // NOLINT

  // clang-format off
  parser.add_argument(
      "integer", nargs="+", choices={1, 2, 3, 4}, dest=&int_args,  // NOLINT
      help="an integer for the accumulator", metavar="N");         // NOLINT

  parser.add_argument(
    "-s", "--sum", action="store_const", dest=&accumulate,  // NOLINT
    const_=sum_fn, default_=max_fn,                         // NOLINT
    help="sum the integers (default: find the max)");       // NOLINT
  // clang-format on

  int parse_result = parser.parse_args(argc, argv);
  switch (parse_result) {
    case argue::PARSE_ABORTED:
      return 0;
    case argue::PARSE_EXCEPTION:
      return 1;
    case argue::PARSE_FINISHED:
      break;
  }

  std::cout << accumulate->GetName() << "(" << string::join(int_args)
            << ") = " << (*accumulate)(int_args) << "\n";
  return 0;
}
```

When executed with `-h` the output is:

```
==========
argue-demo
==========
version: 0.1.3-dev5
author : Josh Bialkowski <josh.bialkowski@gmail.com>
copyright: (C) 2018

argue-demo [-h/--help] [-v/--version] [-s/--sum] <N> [N..]

Flags:
------
-h  --help          print this help message
-v  --version       print version information and exit
-s  --sum           sum the integers (default: find the max)

Positionals:
------------
integer             an integer for the accumulator
                    choices=[1, 2, 3, 4]
```

2.5 Note on designated-initializers

Designated initializers are a C99 feature (as well as an upcoming C++20 feature) that clang interprets correctly (as an extension) when compiling C++, but is not in fact a language feature. The GNU toolchain does not implement this feature. Therefore, while the following is valid when compiling with clang:

```
parser.add_argument("integer", &int_args, {
  .nargs_ = "+",
  .help_ = "an integer for the accumulator",
  .metavar_ = "N"
});
```

We are not allowed to skip any fields in GCC, meaning that if we wish to use designated initializers in GCC, we must use the following:

```
parser.add_argument("integer", &int_args, {
  .action_ = "store",
  .nargs_ = "+",
  .const_ = argue::kNone,
  .default_ = argue::kNone,
  .choices_ = {1, 2, 3, 4},
  .required_ = false,
  .help_ = "an integer for the accumulator",
  .metavar_ = "N",
});
```

Alternatively we could use the more brittle universal initializer syntax with no designators:

```
parser.add_argument("integer", &int_args, {
  /*.action_ =*/ "store",
  /*.nargs_ =*/ "+",
  /*.const_ =*/ argue::kNone,
  /*.default_ =*/ argue::kNone,
  /*.choices_ =*/ {1, 2, 3, 4},
```

(continues on next page)

(continued from previous page)

```
  /*.required_ =*/ false,
  /*.help_ =*/ "an integer for the accumulator",
  /*.metavar_ =*/ "N",
});
```

But this can get pretty tedious. Therefore, unless you're limited to a compiler supporting designated initializers in C++ you may wish to stick to the alternative assignment APIs.

CHAPTER 3

Changelog

3.1 v0.1 series

3.1.1 v0.1.3

dev0:

- On error, usage string is appended to exception message, meaning that subparser usage is printed rather than superparser when subparser is missing a required argument.
- Fix buffer underflow in subparser action when invalid subcommand is used
- Argue –help can now dump JSON instead of text format.
- Argue programs now inherently suports bash autocompletion
- Registering the same action twice will now throw an exception
- AddArgument returns the kwarg object, meaning that fields can be set after the call instead of during (for compilers like GCC which don't support designated initializers.

Closes: 405abc1, 4f5e576, db38521, e95f5f5

dev1:

- Split argue.h/argue.cc into separate files based on concepts
- Remove argue utilities duplicated in util/
- Switch to lower_snake_case method style
- Metadata version number is a string
- Get rid of ARGUE_EMPTY macros

Closes: 0310925, 8d56785

dev2:

- Implement keywords API
- Unify StoreScalar and StoreList into StoreValue
- Assigning to nargs will not replace the action
- If the program user provides a long flag which is not an exact match but is a unique prefix of a known flag, then match that flag.
- Pass column size into get_help() and get_prolog() so that actions can format their own help text.

Closes: 41e5630, 504d241, 73c5d7a, aead983, d5c7d88

dev3:

- Implement debian package build

Closes: 51f1ef7

3.1.2 v0.1.2

- Add support for `--version` and `--help` without the corresponding short flag (i.e. no `-v` or `-h`)
- Add macro shims to work with gcc which doesn't support designated initializers
- Add support for `nargs=REMAINDER`
- Added `argue`/`glog` integration via a function to add command line flags for all the `glog` `gflag` globals
- Did some build system cleanup
- Removed individual exception classes, unifying them into a single one
- Replace hacky assertion stream with `fmt::format()` usages.
- Replace KWargs class with optional containers with KWargs field objects that pass-through to setters instead.
- Don't latch help text at help tree construction time, instead query help out of the action objects at runtime. This is so that subparsers know what children they have and can generate choices text.

3.1.3 v0.1.1

- Implemented subparser support

3.1.4 v0.1.0

Initial release! This is mostly a proof of concept initial implementation. It lacks several features I would consider required but works pretty well to start using it.

Features:

- argparse-like syntax
- type-safe implementations for `store`, `store_const`, `help`, and `version` actions
- support for scalar `(nargs=<default>, nargs='?')` or container (nargs=<n>, nargs='+', nargs='*') assignment

- provides different exception types and error messages for exceptional conditions so they can be distinguised between input errors (user errors), library usage errors (your bugs), or library errors (my bugs).
- support for custom actions
- output formatting for usage, version, and full help text

CHAPTER 4

Release Notes

4.1 v0.1 series

4.1.1 v0.1.3

The library remains still very much in alpha territory. This release includes a couple of exciting new features though. In particular, all programs which use `argue` for argument parsing automatically support:

1. bash completion (no custom completion script necessary!)
2. generated command line documentation using a JSON representation of the command tree

Example program help:

```
==========
argue-demo
==========
version: 0.1.3-dev5
author : Josh Bialkowski <josh.bialkowski@gmail.com>
copyright: (C) 2018

argue-demo [-h/--help] [-v/--version] [-s/--sum] <N> [N..]

Flags:
------
-h  --help          print this help message
-v  --version       print version information and exit
-s  --sum           sum the integers (default: find the max)

Positionals:
------------
integer             an integer for the accumulator
                    choices=[1, 2, 3, 4]
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search