Chapter 12 - Command Line Parsers

You are all most likely familiar with one or more command line parsing libraries for Python. Popular libraries include argparse, optparse, plac and docopt. Argument parsers are vital tools in any developer's kit, allowing us to make short work of complicated user input.

In Python

argparse

Before jumping into C++, lets throw together a simple argparse example to remind ourselves of the common features, shall we ?

First we need to import argparse.

```
#!/usr/bin/env python

import argparse
```

Next we are going to create a function to handle setting up the parser and parsing the command line arguments.

```
def parse():
    parser = argparse.ArgumentParser()
```

And we are going to add a positional argument to the parser. As you might recall, positional arguments come after any optional arguments on the command line.

```
    parser.add_argument('name')
    # add greeting option w/ default
```

Afterwards, we are going to add a couple of flags.

```
    parser.add_argument('--greeting', default='Hello')
    # add a flag (default=False)
    parser.add_argument('--caps', action='store_true')
```

Now we are going to define a function which we wish to be executed upon successful parsing of the arguments.

```
# since we are now passing in the greeting
# the logic has been consolidated to a single greet function
def greet(args):
    output = '{0}, {1}!'.format(args.greeting, args.name)
    if args.caps:
        output = output.upper()
    print(output)
```

And finally, we are going to set a default function to execute, the we are going to actually parse the arguments and execute the function.

```
    parser.set_defaults(func=greet)
    args = parser.parse_args()
    args.func(args)

if __name__ == '__main__':
    parse()
```

Putting it all together, we can execute the script on the command line:

```
>>>parser_eg -h
>>>parser_eg Fred
>>>parser_eg --greeting "Hey dude" Fred
>>>parser_eg --greeting "Hey dude" --caps Fred
```

This is just scratching the surface. Argparse also handles subparsers, as well as a host of other capabilities. You can check out the documentation here:

argparse docs

docopt

Of course, argparse isn't the only game in town. Of particular interest is a relative newcommer to the ecosystem of command line parsers - docopt. Docopt takes a completely novel approach. Rather than have you write commands to build up your parser, Docopt expects you to write posix standard help text, and it parses that help text and generates teh parser. Pretty cool stuff. Here is an example to be clear:

```
"""Naval Fate.

Usage:
  naval_fate.py ship new <name>...
  naval_fate.py ship <name> move <x> <y> [--speed=<kn>]
  naval_fate.py ship shoot <x> <y>
  naval_fate.py mine (set|remove) <x> <y> [--moored | --drifting]
  naval_fate.py (-h | --help)
  naval_fate.py --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Moored (anchored) mine.
  --drifting    Drifting mine.
```

```
"""
from docopt import docopt


if __name__ == '__main__':
    arguments = docopt(__doc__, version='Naval Fate 2.0')
    print(arguments)
```

Sweet eh? Here are the docs:

docopt documentation

What about C++?

As you can imagine, things aren't quite so simple in C++. However, the good news is, that things are **almost** as simple. We will cover a couple parsers and see how to go about doing this. Here is a non-exhaustive list of command line parsers:

- args
- docopt.cpp
- tclap
- boost.Program_options
- cxxopts

I am quite happy with args, so we are going to explore its usage next.

args

One of the best things about args is that it is a header only library. That makes it pretty convenient. Simply download it from github, drop the header in your project, and away you go.

Create a new project

Set up a new project. Download taywee/args from github and place the header file in your project. Be sure to set up your CMakeList.txt file to point to the header file.

We are going to be creating a dummy application which simply parses inputs, for the purpose of illustrating how args works. So, create a `main.cpp` file with a main function and lets get started. . . .

Set up args

First include args into your main.cpp. For me, the include line is:

```
#include "args/args.hxx"
```

Our first flag - the Help flag

Next, lets jump to our main function. We are going to walk through creating a couple of different types of flags and positional arguments to learn about args.

Lets start off by creating an args parser. The ArgumentParser takes a couple of arguments. The first is the usage string, and the second is a string which is presented after the OPTIONS.

```
    args::ArgumentParser parser("This is a test program.", "This goes after the options.");
```

Next, we are going to add the help flag. This flag will either be active or not, but it wont take any arguments.
To do so, we are going to use the Help class provided by args. It takes a couple of arguments:

1. The parser or group to which the flag applies. We have not talked about groups yet, but we will.
2. The name of the argument as a string
3. The help string
4. a pair of single character and string names for the short and long flags. This is initialized in mustaches. {}. Note the single quote for the character, and the double quote for the string.

```
    args::HelpFlag help(parser, "help", "Display this help menu", {'h', "help"});
```

parsing

Ok. Lets add the code to actually parse the inputs. There is a bit of boilerplate here. First, we are going to use a try catch block, as args throws exceptions by design. The first thing you will notice is that args throws an exception if it encounters the help flag. That clears up a mystery that the observant among you might have noticed (like how does args know anything about help? All we did was add a help flag. Well it is special).

Below testing for help, we check to see if there is a ParseError or a ValidationError. One indicates that there is an error with the parser itself, and the other indicates that there was a logical error with the inputs. Which is which? Don't recall off the top of my head. See the docs.

```
try
    {
        parser.ParseCLI(argc, argv);
    }
```

```
catch (args::Help)
{
    std::cout << parser;
    return 0;
}
catch (args::ParseError e)
{
    std::cerr << e.what() << std::endl;
    std::cerr << parser;
    return 1;
}
catch (args::ValidationError e)
{
    std::cerr << e.what() << std::endl;
    std::cerr << parser;
    return 1;
}
```

Adding a boolean flag

The simplest flag, short of the Help flag, is the `args::Flag`. It takes a couple of arguments:

- a parser instance (or group instance)
- the flag name as a string
- a description
- the flags as a list surrounded by {}.

```
args::Flag bar(parser, "bar", "are we going to get a bar?", {'b', "bar"});
```

Adding a flag which takes an argument.

Ok, lets get adventurous and add a flag that takes a string. The class is almost identical to what we have seen so far. We use a template class to add flags which take values. The class is called ....`args::ValueFlag`.

```
args::ValueFlag<std::string> name(parser, "name", "a name", {'n',"name"});
```

Oh, and since we are taking a string, go ahead and add an appropriate #include. I'll wait...

Adding a flag which takes a list of arguments

If that isn't cool enough, args has a mechanism for defining a flag which may be used multiple times. The name is, appropriately enough, `args::ValueFlagList` and it is also a template function:

```
args::ValueFlagList<std::string> friends(parser, "friends", "friends of the dude", {'f'
```

Positional arguments

Lets add a positional argument. Doing so is as simple as using the `args::Positional` template class.

```
args::Positional<std::string> exclamation(parser, "exclamation", "the exclamation uttere
```

Positional argument List

There is also a variant of args::Positional which takes a list of arguments. It uses a template class called `atgs::PositionalList` which is used thusly:

```
args::PositionalList<double> numbers(parser, "numbers", "The numbers position list");
```

Evaluating the results.

Ok, we know how to define basic flags and positionals, but how do we evaluate the results? Well, this is actually quite straightforward. Lets jump to back to main, after our try catch block, and test for our args:

In general, you simply test the class instance of the flag which you created, and, if it is true, you use the `args::get` method to fetch the results.

```
if (name) { std::cout << "name : " << args::get(name) << std::endl; }
if (friends) { for (const auto fr: args::get(friends)) { std::cout << "c: " << fr << " '
if (bar) { std::cout << "bar was active" <<std::endl; }
if (exclamation) { std::cout << args::get(exclamation) << std::endl; } }
if (numbers) { for (auto& n : args::get(numbers)) {std::cout << n << " ";} std::cout <<
```

Args Groups

Ok, now that we have the basics down, lets look at a more sophisticated feature of args - groups. Args allows you to group flags and positional arguments together to create more complex systems. For argument's sake, lets say that we want to accept one of two flags, foo and bar, and/or one of two other flags, baz and bla. How would we go about this? Groups!

```
args::Group group(parser, "This group is all exclusive:", args::Group::Validators::AtMos
args::Flag foo(group, "foo", "The foo flag", {'f', "foo"});
args::Flag bar(group, "bar", "The bar flag", {'b'});

args::Group group2(parser, "This group is also exclusive:", args::Group::Validators::AtM
args::Flag baz(group2, "baz", "The baz flag", {"baz"});
args::Flag bla(group2, "bla", "The bla flag", {"bla"});
```

There are many validators provided by args, and you can even write your own. If you are interested, search through the source for Group to see more...

Extra Credit - Sup parsers

Admitedly, this is an area of some complication in args. If you want to write the next git, with a large number of sub commands, you are in luck. Arg has you covered. However, I wouldn't blame you for skipping ahead to the next section.

For the intrepid few looking for a challenge, first you need to define a function signature via std::function:

```
using commandtype = std::function<void(const std::string &,
                                       std::vector<std::string>::const_iterator,
                                       std::vector<std::string>::const_iterator)>;
```

This type represents a sup-parser. We need to define one or more sub parser functions which have the commandtype signature. Beacuse we will need to register them with args, we forward declare them. For example:

```
// forward declarations
void ShelfFn(const std::string &progname,
             std::vector<std::string>::const_iterator beginargs,
             std::vector<std::string>::const_iterator endargs);

void PluginFn(const std::string &progname,
              std::vector<std::string>::const_iterator beginargs,
              std::vector<std::string>::const_iterator endargs);
```

Next, we need to set up the parser. I am doing this in the main function. First, I create a map mapping name to function:

```
    std::unordered_map<std::string, commandtype> map{
            {"shelf", ShelfFn},
            {"plugin", PluginFn}};
```

Next we create a vector of strings and initialize them with all of the arguments to main ,excluding the zeroth argument, which is the application name

```
    const std::vector<std::string> args(argv + 1, argv + argc);
```

Next, we create the argument parser, add a lehp flag, set the program name.

```
    args::ArgumentParser parser("resman", "COMMAND:  shelf | plugin\n\n");
    args::HelpFlag help(parser, "help", "Display this help menu", {'h', "help"});
    parser.Prog(argv[0]);
    parser.ProglinePostfix("{command options}");
```

Finally, we register the sub commands with args via the MapPositional template class, and evaluate the results

```
    args::MapPositional<std::string, commandtype> command(parser, "command", "Command to exe
    command.KickOut(true);
    try
```

```
    {
        auto next = parser.ParseArgs(args);

        if (command)
        {
            args::get(command)(argv[0], next, std::end(args));
        } else
        {
            std::cout << parser;
        }
    }
    catch (args::Help)
    {
        std::cout << parser;
        return 0;
    }
    catch (args::Error e)
    {
        std::cerr << e.what() << std::endl;
        std::cerr << parser;
        return 1;
    }
    return 0;
}
```

Finally, we have to define the functions we forward declared. You will notice that each function houses a full blown parser in its own right:

```
void ShelfFn(const std::string &progname,
           std::vector<std::string>::const_iterator beginargs,
           std::vector<std::string>::const_iterator endargs) {
    args::ArgumentParser parser("");
    parser.Prog(progname + " shelf");
    args::HelpFlag help(parser, "help", "Display this help menu", {'h', "help"});

    args::ValueFlag<std::string> output(parser, "output", "The path to the directory in whic
                                        {'o',"output"});
    args::Positional<std::string> shelf(parser,"shelf", "The path to the shelf mel file");

    try {
        parser.ParseArgs(beginargs, endargs);
        if(output and shelf) {
            extractShelves(args::get(shelf), args::get(output));
        } else if(shelf) {
            printShelf(args::get(shelf));
        } else {
```

```
            std::cout << parser;
        }

    } catch (args::Help) {
        std::cout << parser;
        return;
    } catch (args::ParseError e) {
        std::cerr << e.what() << std::endl;
        std::cerr << parser;
        return;
    }catch(std::exception& e) {
        std::cerr << e.what() << std::endl;
        return;
    } catch(...) {
        std::cerr << "problem parsing input" << std::endl;
        return ;
    }

}


void PluginFn(const std::string &progname,
            std::vector<std::string>::const_iterator beginargs,
            std::vector<std::string>::const_iterator endargs)
{
    args::ArgumentParser parser("");
    parser.Prog(progname + " plugin");
    args::HelpFlag help(parser, "help", "Display this help menu", {'h', "help"});
    args::Flag validate(parser, "validate", "validate input argument", {'v', "validate"});
    args::ValueFlag<std::string> dcc(parser, "dcc", "The applicable dcc: maya, houdini, nuke
                                     {'d',"dcc"});
    args::ValueFlag<std::string> output(parser, "output", "save the output to a file", {'o',
    args::Positional<std::string> plugin(parser, "plugin", "the name of the plugin to load")

    try
    {
        parser.ParseArgs(beginargs, endargs);
        std::string dcc_str{"maya"};
        if (dcc)
            dcc_str = args::get(dcc);

        if(dcc_str != "maya") {
            std::cerr << "\"{}\" is not a supported dcc. Currently, only \"maya\" is support
            return;
        }
        if(plugin) {
```

```
            auto plugin_str = args::get(plugin);

            std::string output_str = "resmanLoadPlugin(\"{}\");\n"_format(plugin_str);
            //
            // Validate provided plugin string, assuring that it exists.
            //
            if(validate) {
                fs::path opath{plugin_str};
                if(!fs::exists(opath)) {
                    std::cerr << "Error. The provided path: \"{}\" does not exist.\n"_format
                    return;
                }
            }
            //
            // Output plugin loader to file
            //
            if(output) {
                fs::path opath{args::get(output)};
                if(!opath.has_filename()) {
                    std::cerr << "error: {} is not a valid output path"_format(args::get(out
                    return;
                }
                // make sure that parent path exists
                auto parent_path = opath.parent_path();
                if(!fs::exists(parent_path)) {
                    std::cerr << "Error: The provided path: \"{}\" does not exist."_format(p
                    return;
                }
                // make certain there is a "mel" extension.
                opath.replace_extension("mel");

                std::ofstream out(opath.string());
                if(!out) {
                    std::cerr << "Error: Unable to open \"{}\" for writing.\n"_format(opath.
                    return;
                }
                out << output_str;
                out.close();
            } else {
                std::cout << output_str << std::endl;
            }
        } else {
            std::cout << parser;
            return;
        }
    } catch (args::Help) {
```

```
        std::cout << parser;
        return;
    } catch (args::ParseError e) {
        std::cerr << e.what() << std::endl;
        std::cerr << parser;
        return;
    } catch(std::exception& e) {
        std::cerr << e.what() << std::endl;
        std::cerr << parser;
        return;
    }
}
```

docopt in C++

Docopt works almost exaclty the same in c++ as it does in python.

```
#include "docopt.h"

#include <iostream>

static const char USAGE[] =
R"(Naval Fate.

    Usage:
      naval_fate ship new <name>...
      naval_fate ship <name> move <x> <y> [--speed=<kn>]
      naval_fate ship shoot <x> <y>
      naval_fate mine (set|remove) <x> <y> [--moored | --drifting]
      naval_fate (-h | --help)
      naval_fate --version

    Options:
      -h --help     Show this screen.
      --version     Show version.
      --speed=<kn>  Speed in knots [default: 10].
      --moored      Moored (anchored) mine.
      --drifting    Drifting mine.
)";

int main(int argc, const char** argv)
{
    std::map<std::string, docopt::value> args
        = docopt::docopt(USAGE,
                         { argv + 1, argv + argc },
```

```
                        true,               // show help if requested
                        "Naval Fate 2.0");  // version string

    for(auto const& arg : args) {
        std::cout << arg.first <<  arg.second << std::endl;
    }

    return 0;
}
```

Unlike args, docopt is not a header only library. However, it is trivial to build:

- git clone repo
- cd into repo
- create a build directory
- cd into the build directory
- type cmake ..
- type make

After that you can either move the shared and/or static library and headers someplace meaningful to you manually, or use make install.

There is one caveat with docopt which we need to go over: it needs a pretty recent compiler version to work:

- Clang 3.3 and later
- GCC 4.9
- Visual C++ 2015 RC

Homework

Pick one of the c++ option parsers mentioned but not illustrated and get it to work. Parse the shait out a command line by next week!