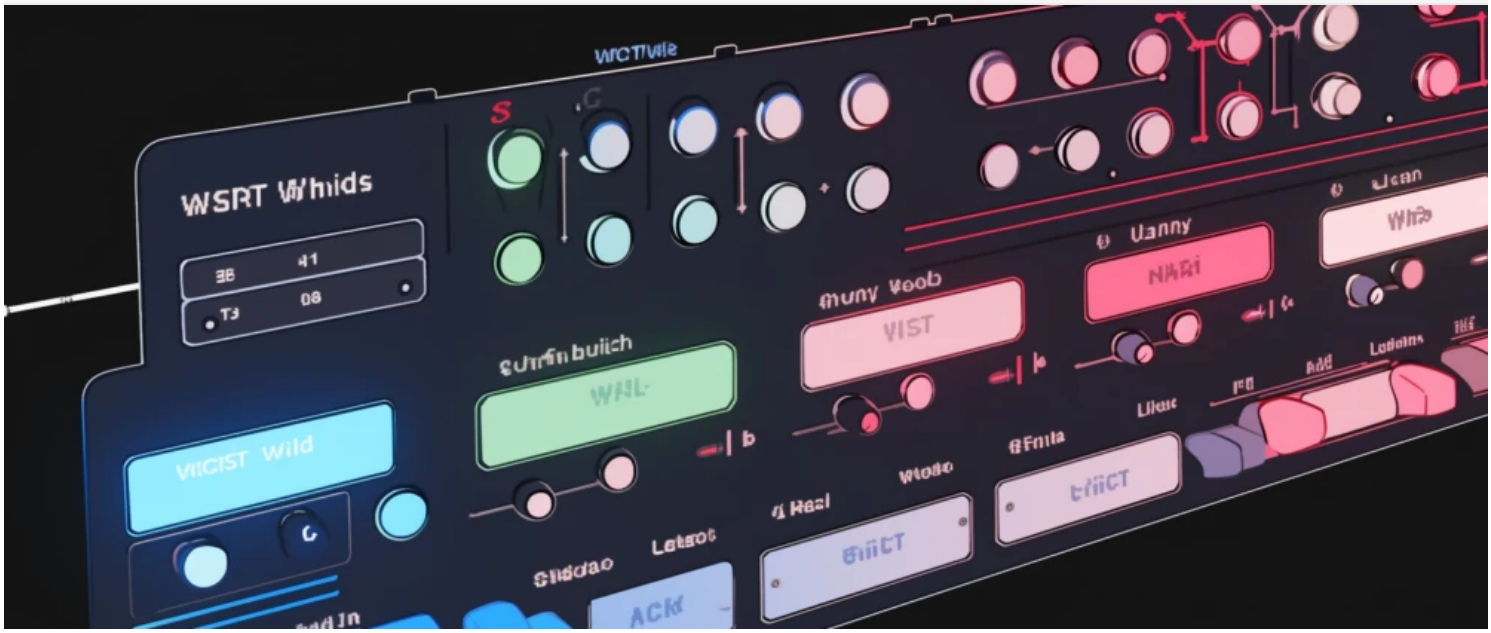

Remo Dentato
Posted on Sep 21, 2023 • Updated on Oct 1, 2023

Easy command line interfaces in C

#c #cli #library

To C and Beyond (13 Part Series)

1. Exceptional C
2. Default parameters in C
3. **9 more parts...**
12. What's in a C (NaN)box?
13. A Simple Unit Test Framework for C

Overview

I already introduced the VRG library describing the support it provides for writing variadic functions in C. Here I'll discuss the support it provides to create command line interfaces for C programs.

The code is freely available (under the MIT Licence) library is available on Github.

Command-line interfaces (CLIs) have always been a fundamental aspect of software development, especially in C programming, where simplicity and control are highly valued. Traditionally, CLI handling in C has been a tedious chore, often filled with repetitive, error-prone code. Existing solutions like the GNU `getopt` library provide a robust feature set but can be somewhat complex to integrate and use.

Simplicity is Key

One of the standout features of VRG is its simplicity. It offers a clean and intuitive API that reduces boilerplate code and lets you focus on what really matters—your application logic. The VRG library abstracts away the cumbersome parts of CLI handling and provides you with an easy-to-use interface that's both developer-friendly and efficient.

For example, it eliminates the need to write (and maintain!) a separate `usage()` function with the description of arguments and flags. Also, it offers an intuitive way to mark optional arguments.

Why Choose VRG Over GNU Getopt?

GNU `getopt` is certainly a robust library, but its API can be daunting for beginners or even intermediate programmers. With `getopt`, you often find yourself managing multiple global variables, configuring long and short options separately, and worrying about the sequencing of options and arguments.

VRG, on the other hand, provides a more intuitive and straightforward approach to CLI parsing. With a clean, high-level abstraction, you can define flags, their types, and the actions to perform when they are encountered—all enclosed within a single `vrgcli()` function. It simplifies the entire process and does away with the complexities involved in using something like `getopt`.

Getting Started with VRG

Using VRG is as simple as including the header file and defining `VRGMAIN` in one of your source files:

```
#define VRGMAIN
#include "vrg.h"
```

After that, you can directly jump into defining your CLI options and arguments using `vrgcli()` and `vrgarg()` functions. Here's a quick example to demonstrate its usage:

```
vrgcli("MyProgram v1.0 (c) 2023 by Me") {
  vrgarg("-h\tDisplay help") {
    vrgusage();
  }
  vrgarg("-f [filename]\tSpecify an optional filename") {
    printf("Filename: %s\n", vrgarg);
  }
}
```

How Does It Work?

The `vrgcli()` function acts as a container for all your CLI options and arguments. Each `vrgarg()` block within `vrgcli()` defines a specific flag or argument, along with the code to execute when that flag or argument is encountered.

The flag and its description are separated by a tab character `\t`. The part before the tab is the actual flag specification (`-f [filename]`), and the part after the tab is the description that will be displayed when you call `vrgusage()`.

Inside the `vrgarg()` block, you have access to a `char` pointer `vrgarg` that points to the argument's value, making it easy to handle the parsed data.

Show Me the Code

Here's a complete example that demonstrates the simplicity and efficiency of VRG:

```
#define VRGMAIN
#include "vrg.h"

int main(int argc, char *argv[]) {
  vrgcli("My Awesome CLI Program v1.0") {
    vrgarg("-h\tDisplay help") {
```

```
      vrgusage();
    }
    vrgarg("-f [filename]\tSpecify an optional filename") {
      printf("Filename: %s\n", vrgarg);
    }
    vrgarg("input\tInput file") {
      printf("Processing input file: %s\n", vrgarg);
    }
    vrgarg() {
      vrgusage("Unexpected argument: '%s'\n", vrgarg);
    }
  }
  return 0;
}
```

As you can see, VRG allows you to set up a fully functional CLI with just a few lines of code!

The `usage()` function is automatically generated and mandatory arguments are checked:

```
$ gcc -o myprog myprog.c
$ ./myprog
Error: missing 'input' argument
USAGE:
  myprog [OPTIONS] input ...
  My Awesome CLI Program v1.0

ARGUMENTS:
  input         Input file

OPTIONS:
  -h            Display help
  -f [filename] Specify an optional filename
```

Flags and positional arguments can be intermixed according the user's preference so that the two lines below are equivalent:

```
  myprog -f pluto pippo
  myprog pippo -f pluto
```

As added flexibility, the user can specify flags arguments together with the flag itself or as a separate argument (`-f pluto` is fully equivalent to `-fpluto`).

Flags can be specified in a *long* or *short* format. For example:

```
myprog -f pluto pippo
myprog --file=pluto pippo
```

Definining them in this way:

```
vrgarg("-f, --file filename\tThe file to consider") {
  // ...
}
```

Custom Validators

One of the powerful features of VRG is the ability to add custom validation functions. This functionality allows you to set specific rules for the values that command-line arguments can take, offering an additional layer of robustness to your application. What's more, these validation functions can also take additional parameters for more nuanced control.

Quick Example: File Readability Check

Imagine you need to ensure that an argument specifies a file that exists and is readable. You can define a custom validator function like this:

```
// Check if the specified file exists and is readable
int isfile(char *arg) {
  if (arg == NULL || *arg == '\0') return 0;
  FILE *f = fopen(arg, "rb");
  if (f == NULL) return 0;
  fclose(f);
  return 1;
}
```

And then you can use it in your `vrgarg()` function:

```
vrgarg("-f file\tSpecify the input file", isfile) {
  // Your code here, safely reading the file
}
```

This ensures that an error will be thrown if the user attempts to specify a file that does not exist or is not readable, thereby enhancing the resilience and user-friendliness of your CLI.

Incorporating custom validation logic is just another way VRG offers you more control while maintaining its simplicity.

Speaking Your Audience's Language

I'm convinced that CLI should be in English whenever possible. This would enlarge the number of potential users since English is the "de facto" standard in many technical environment.

However, it may be appropriate sometimes to localize the CLI to enhance the user experience. If you're catering to a non-English speaking audience, localized error messages can make your software more user-friendly and reduce potential user friction.

For instance, if your target audience is primarily Japanese, VRG allows you to redefine internal message strings for a Japanese interface:

```
// re-define the internal messages for Japanese
// Note: These translations are machine-generated.

#define VRG_STR_USAGE        "使い方"
#define VRG_STR_ERROR        "エラー"
#define VRG_STR_INVALID      "%T '%N' に対する無効な値 '%V'"
#define VRG_STR_INV_OPTION   "オプション"
#define VRG_STR_INV_ARGUMENT "引数"
#define VRG_STR_NO_ARGOPT    "オプション '%.*s' の引数が不足しています"
#define VRG_STR_NO_ARGUMENT  "引数 '%.*s' が不足しています"
#define VRG_STR_OPTIONS      "オプション"
#define VRG_STR_ARGUMENTS    "引数"

#include "vrg.h"
```

With these redefinitions, and the proper `vrgarg()`s, the CLI's help output might resemble:

```
使い方:
  vrgtest4 [オプション] モデル [出力ファイル名] ...
```

```
  バージョン: 1.3RC
  vrgli 関数 の デモ

引数:
  モデル                          モデル の ファイル 名
  [出力ファイル名]                生成するファイル

オプション:
  -h, --ヘルプ                     この ヘルプ を表示
  -n, --数-光線 数                 光線 の 数（正 の 整数）
  -t, --なぞる [はい/いいえ]       輪郭をなぞる（ブーリアン）
  --練習
  -r                               再びなぞる
```

Apart from internationalization, this customization feature is invaluable when the default error messages are not detailed enough for your specific use case. Tailoring these strings allows developers to provide feedback that is more contextually relevant, helping users understand exactly what went wrong and potentially how to rectify it.

VRG's capability to redefine error messages and usage strings ensures that developers can craft CLI experiences that are intuitive, relevant, and culturally appropriate. Whether you're prioritizing internationalization or domain-specific feedback, VRG equips you with the tools to communicate effectively with your users.

Conclusion

VRG offers a fresh and straightforward approach to CLI parsing in C. With its clean API and simple syntax, you can define complex command-line interfaces without diving into the intricacies of traditional parsing libraries like GNU `getopt`. If you're a C developer looking for a simpler and more efficient way to handle CLIs, give VRG a try. It might just be the tool you never knew you needed.

To C and Beyond (13 Part Series)

1 Exceptional C

2 Default parameters in C

... **9 more parts...**

Top comments (15)

Paul J. Lucas • Sep 21 '23

Yes, long-option support is a must, especially for programs that have lots of options (for example, my own wrap). While I personally always have a short-option equivalent for every long option, eventually you end up with collisions and have to use letters that aren't mnemonic — which is why having a long option helps the user. Additionally, *all* programs should accept the `--help` and `--version` options.

This may fall into personal taste, but when the user errs (e.g., forgets a required argument for an option), it's best to print an error message *only* about that and *not* a full dump of the complete usage leaving the user to have to scan a large output looking for the relevant portion of the usage for just that option.

Ideally, a robust CLI library should also have mechanisms for:

- Being able to specify validator functions that parse and validate option arguments.
- Being able to specify which options are mutually exclusive from all other options (e.g., if you specify `--help`, you may not specify any other option or program arguments).
- Being able to specify which options are mutually exclusive with other options.
- Being able to specify the same option more than once, for example `-v` for "verbose" output, but `-vv` for *more* verbose output. (Though you probably can support this now by having your handler code increment a counter.)

But I do agree that GNU's `getopt_long()` leaves a lot to be desired.

Remo Dentato • Sep 21 '23

Thanks for your comment! I'll think about long args.
Validation (including dependency between options) seems more complicated to add while keeping the current level of simplicity.
For example passing a `check` function `vrgarg`? SOmething like: `vrgarg("-x\tXmas",check)`?

Paul J. Lucas • Sep 21 '23

It *may* be possible (paraphrasing Alan Kay) to have simple CLIs be simple, but complex CLIs be possible. I think it would be a neat trick if you could manage to do it all via a header-only library, but I think it's unlikely.

The GNU `struct option` ideally needs more fields. As for `vrgarg`, a `_Bool check(char const*)` function would be a start. You could have the library provide a few built-in checkers like `vrgarg_check_int`, `vrgard_check_double`, etc., but the user would be free to supply their own. The check function should be responsible for printing any error message since only it knows what the correct format is. The `_Bool` return value would only indicate to the library that a `false` value means "stop processing" and `exit(EX_USAGE)`.

Remo Dentato • Sep 21 '23 • Edited

As you (and Alan Kay) said, *It should be possible* :)
It was just a dozen of lines of code, so I added the ability to specify a custom validator for each argument (possibly using additional parameters).
Have a look at the code on Github, or, if you feel inclined to do so, join the Discord server I just created to talk about `vrg`.

With this custom validators it should be quite easy to implement a logic of mutual exclusion between flags.

I'm not sure that providing pre-made validators would be useful, they would probably ending up being too simple and generic (and if this is what is needed, the users might write the validator themself).

Paul J. Lucas • Sep 22 '23

IMHO, it's a bad idea to put full-blown function definitions into a header file, even if they are declared `static`.

It's also not clear why you're using `char` where `_Bool` is more appropriate. Similarly, for `vrg_def_s:: hasarg`, you're assuming `char` is signed — which it's not guaranteed to be. IMHO, you should use an `enum`, but if you insist on using signed integers, at least use `signed char`.

In general, the varargs stuff and the specific use-case for CLI arguments are too intertwined. If you want to make a varargs library, fine; if you want to make a CLI library, fine; but don't conflate them.

Remo Dentato • Sep 23 '23 • Edited

Thanks for reminding me about the `char` signedness. I completely forgot.
I did not include `<stdbool.h>` as this would force anyone including "vrg.h" to have symbols like `true` defined even if they do not want to.
Yes, I might enforce that anyone should abide at least to the C99 standard but I can't see a benefit for me to put such a restriction.
Considering that the CLI functions will only be used where `main()` is, I thought it would be more convenient to ask the user to define a symbol before including "vrg.h" rather than having both to include "vrg.h" and link against a "vrg.o" object file. I might considering providing options for both usage. I need to think about it.
As for "mixing" things, I do see the topic of handling varidic "arguments" for functions and parsing arguments for CLI very related. But maybe it's just me, I'd love to hear from others. In any case, since I want to provide a flexible interface, I would need to define the variadic function piece anyway.
Thanks for your comments, I'm now adding the long options as you

suggested and I will probably add a way to specify a full "usage string" because I recognize that the automatically information generated by `vrgusage()` might be not enough. Imagine you want to provide the help In other languages than English ...

Paul J. Lucas • Sep 26 '23

I never said you needed to `#include <stdbool.h>` nor use `true` or `false`; I said to use `_Bool` which is a type built into C99.

> Yes, I might enforce that anyone should abide at least to the C99 standard but I can't see a benefit for me to put such a restriction.

C99 is 24 years old. I think it's perfectly fine to assume that compilers support it.

Remo Dentato • Sep 26 '23

My fault, I always see using `_Bool` and including `stdbool.h` as a single thing.
In any case, looking at the current code, you'll notice that now that field is used to hold a set of bit flags, not just as boolean (actually this always was the case, the old name suggested it was a simple boolean, that's way I changed it).
Now, it could make sense to have `vrg_argfound` defined as _Bool but I don't see the need.
Rather, I would be very interested in your comments on the way the CLI is defined. I saw your article and your `wrap` repo.
The way the CLI is defined, the short/long options are specified, etc, do they seem simple to you? Or, on the contrary, they look too confusing? WHat do you think?

Paul J. Lucas • Sep 26 '23

If you're using bit flags, you should definitely use an `unsigned` type; or a type like `uint8_t`.

I haven't given a lot of thought to how I would write an option parser, but I'd definitely have a distinct object for each option, then an array of pointers to option to pass to a function to parse all options. I don't see why varargs are needed at all.

Remo Dentato • Sep 26 '23 • Edited

Yes, it is an `unsigned` type, in fact.
The variadic functions are needed because I wanted to offer a flexible interface. One can define an option simply as:

```
vararg("-n, --nodes num\tSet the number of nodes") {
}
```

or may want to add a custom validator (was one of your suggestions, right?):

```
vararg("-n, --nodes num\tSet the number of nodes", isgreaterthan
}
```

Similarly, for `vrgcli()` you can omit the `argc` and `argv` arguments if they are the idiomatic ones.
I wanted the functions to be simple to use yet flexible.

Under the hood, the argument definitions are kept in a linked list rather than an array.

Paul J. Lucas • Sep 27 '23

The general problem with varargs is they're not type-safe. I'd prefer always providing a validator even if it's a pre-defined do-nothing one (for string values).

Remo Dentato • Sep 27 '23 • Edited

I'm not using stdarg.h. The technique I use enforces type checking like any regular call.
Validators can have additional arguments (like in the example I used `isgreaterthen, 0`). Yes you can have different validators (`isgreaterthan20`, `islowerthan31`, ...) but I thought that giving the ability to specify additional parameters would have been easier for the programmer.
Can you suggest a set of validators you think would be generally useful to have out of the box?

Remo Dentato • Sep 26 '23

I'm going to add the possibility of translating the various generated messages so to allow the creation of CLI in languages other than English.

I really don't like translated CLI (and I'm not a native English speaker) but I understand this might be a need for certain types of tools.
What do you think?

Remo Dentato • Sep 24 '23

UPDATE:
Just pushed a new version with support for long options and modified the article accordingly.
Thanks for your feedback. I'll be happy to hear any comment you may have.

Remo Dentato • Sep 27 '23

UPDATE: vrg now allows the configuration the user messages. An example for Japanese has been added to the code on GH

Remo Dentato

I'm an old-time programmer for work and for fun. C is my favorite color. I'd love to share some thoughts on programming that go beyond the basics.

LOCATION
Italy

WORK
Dev

JOINED
Dec 11, 2019

More from Remo Dentato

A Simple Unit Test Framework for C
#c #unittest #testing

What's in a C (NaN)box?
#c #nanboxing

Express yourself (in C)
#c #macro

Official tag for Facebook's React JavaScript library for building user interfaces

#python Follow

import antigravity

#devops Follow

Content centering around the shifting left of responsibility, deconstruction of responsibility silos, and the automation of repetitive work tasks.

Explore Our Yearly Summary