Proposal for C2y

WG14 N2948

Accessing the command line arguments outside of main()

Reply-to:

Corentin Jabot (corentinjabot@gmail.com)

Aaron Ballman (aaron@aaronballman.com)

Document No: N2948
Date: 2022-03-17

Abstract

We propose to add the following two functions to be able to query the command line arguments anywhere in a program:

```
#include <stdlib.h>
int _Get_argc(void);
const char * const *_Get_argv(void);
```

Motivation

Many higher level languages and frameworks attempt to expose the program arguments outside of main. Indeed, maybe they want to expose some kind of command line parser facility, or a view of the function programs that take into account the complexity of text encoding.

Swift

The CommandLine class can be used as follows:

```
var c = 0;
for arg in CommandLine.arguments {
    print("argument \(c) is: \(arg)")
    c += 1
}
```

Note that the CommandLine entity can be used anywhere in the program.
To achieve this interface, Swift uses

- Parsing `/proc/self/cmdline` on Linux systems, which causes the Swift runtime to allocate memory for the parameters, even if these parameters are already in the C runtime's memory.
- Calling `_NSGetArgc` and `_NSGetArgv` on OSX, which are undocumented APIS
- Calling `CommandLineToArgvW` on Windows, which again, creates extra allocations
- use `KERN_PROC_ARGS` on FreeBSD

The code can be found here.

Rust

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

Rust uses some of the same tricks as Swift but also

- Uses `NSProcessInfo` on iOS
- On Posix systems, Injects a callback in `init_array` in order to collect the program arguments.

The code can be found here.

WG21

In C++, there have been musings to improve the interactions with command line arguments in main (P0781) and in the entire program (P1275). We note that the shape of main's API forces teachers to expose students to pointers very early in a course/training, which is not always desirable.

Qt

Even if the `QCoreApplication` is globally available after construction, it must be constructed with `argc/argv`.

```
void f() {
   auto args = qApp->arguments();
}

int main(int argc, char** argv) {
   QCoreApplication app(argc, argv);
   f();
}
```

Other use cases

C++ test frameworks all require to be initialized with command line arguments, so they usually define their own main, or require special handling in main.

C frameworks

Some C Frameworks are required to be initialized with command line arguments:

- GTK: `void gtk_init (int* argc,  char*** argv)`.
- GLUT: void glutInit(int *argcp, char **argv);
- Iup: `int IupOpen(int *argc, char ***argv)`

We can observe that the lack of standard solutions for accessing the command line arguments outside of main forces languages that depend on C and C frameworks to resort to non-portable, non-optimal solutions.

We also note that this is a question that comes up regularly on internet forums.

Existing Practice and Implementations

This proposal is inspired by:

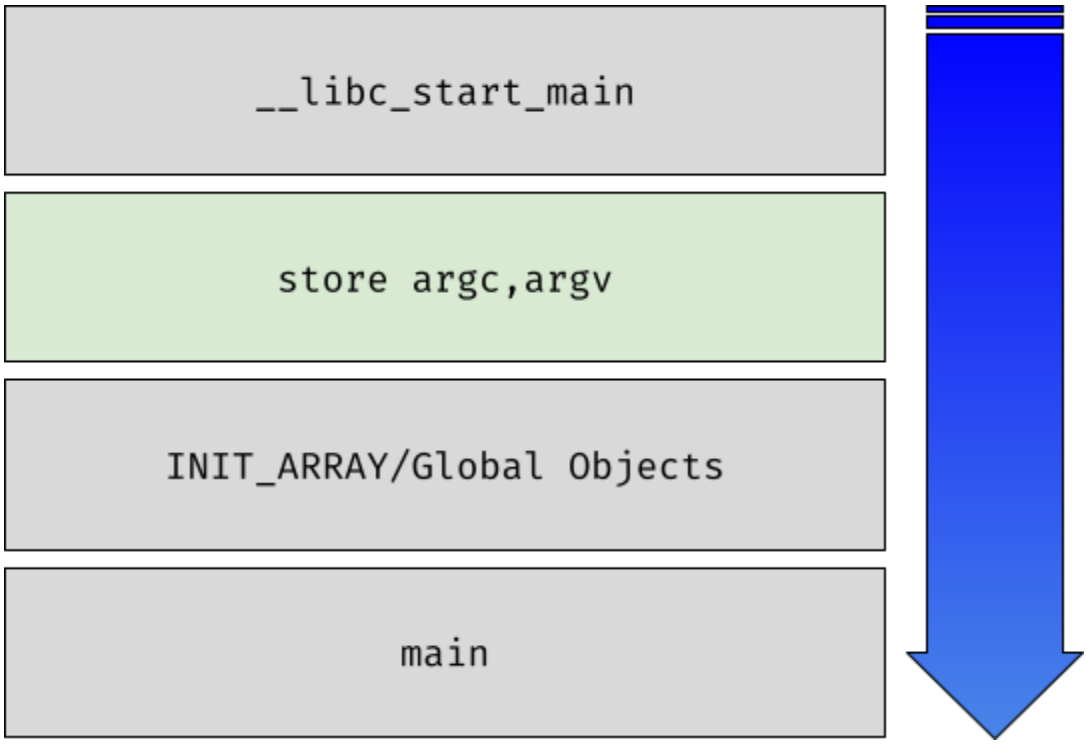
- `__argc, __argv, __wargv` in the C runtime on windows
- `char ***_NSGetArgv(void)` and `int *_NSGetArgc(void)` in the OSX Libc
- __environ in the POSIX spec is based on the same model https://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap08.html
- `NSProcessInfo::arguments` in OSX
- `GetCommandLineW` in win32

In addition, the proposal as proposed was implemented in a private fork of glibc.

Design

Potential implementation strategy

argv's lifetime is already specified to last until the program termination, and exists in main's caller. An implementation needs only to expose that state globally, early in `__libc_start_main` (before the initialization of global variables), so that it is accessible to global objects.



Note that depending on the implementation, the proposed functions may not be callable during the initialization of shared libraries.

This proposal entails storing and assigning a pointer object and an int object. On some implementations (glibc for example), the global state may already exist and so this proposal has no cost at all.

On the other hand, implementations that never call main - such as windows using WinMain or `wmain` may have to do extra work to make this information available, but could decide to return `0`/a pointer to an empty string in that non-conforming scenario.

Is this proposing bad global access that would lead to less testable code, or vulnerability issues?

As shown above, the command line arguments are globally accessible on many systems, and these accesses are necessary for the behavior of widely deployed and programming languages. As such, we are not proposing a new capability, but rather to replace the myriad of workarounds deployed to get to this information with a portablem standard mechanism.

Command line arguments are no different in nature than environment variables. They are part of the environment.

Libraries which need to support mocking or other debugging techniques can keep offering an interface taking argc/argv explicitly and as such the proposed facility does not negatively impact testing or composition.

Command line arguments are globally available in

- `Fortran (get_command_argument/command_argument_count)`
- `Haskell (getArgs)`
- `Ada (Argument_Count/Argument in Ada.Command_Line)`
- `Rust (std::env::args)`
- `Swift (CommandLine.arguments)`
- `Go (os.Args)`
- `Python (sys.args)`
- `Julia (ARGS)`
- `C# (Environment.GetCommandLineArgs())`
- `Nim (commandLineParams)`
- `Zig (std.process.args)`
- `Perl (@ARGV)`
- …

Const correctness and thread safety

The proposed `_Get_argv` returns a `const char* const*`. Globally accessible mutable variables are more likely to introduce bugs, security issues, and general brittleness. They also require more care in a multi-thread program. constness is much safer.
However, and rather regrettably, it is possible to get a mutable pointer to `argv` from `main()`. There is a difference in semantics depending on whether these functions return mutable values or not.

If the value returned by `_Get_argv` is const, then what is returned are the command line arguments, aka a property of the program's immediate environment, and _Get_argv is then a function which queries that environment.

However, if the value is mutable, then... it's just an unreliable global state that can be mutated by anything in the program, at which point it loses in value and fails to say what's on the tin: provide the command line arguments.

Worse, argv could be modified before main!

Alas, our choices are limited. `main()` can mutate its arguments. And **we are not proposing to change that as it affects existing code.** So we take the position that argv as always been mutable, and users do rely on that for a few use cases including:

- Hiding from users args that are specific to a given framework
- Changing the name of the program as displayed by process monitoring tools.

On the other hand, maybe argv should have never been mutable to begin with and we should not perpetuate past mistakes: Such a design was great for small, single threaded programs with no dependencies, but caused maintenance issues and security concerns in large, multi-threaded applications relying on third-party libraries.

So we recommend:

- Returning immutable values from `_Get_argv` (for both the array and the contained strings)
- Keep letting users mutate `argv` from main if they need, for backward compatibility.
- Considering whether we should discourage that practice.

Which, to be perfectly clear, is a compromise rather than a perfect solution: an application may still need to protect concurrent access to _Get_argv if there is a risk that `argv` is modified.

API Shape

With the exception of constness as mentioned above, the proposed functions model exactly the types and values of `argc` and `argv` as passed in main.
This is intentional: We do not want to force implementations to do further transformations (especially expensive transformations), nor do we want to offer an unfamiliar API.

Naming/Header/Freestanding

We would prefer this facility be in stdlib.h, as that is where `getenv` and other environment-related functions are. Because the notion of command line arguments, or even main() does not always make sense in a freestanding environment, a conforming freestanding implementation does not have to provide these functions.

We propose `_Get_argv`() / `_Get_argc`() for the name of these new facilities.

Can an implementation simply return 0/a pointer to an empty string?

Yes, the intent is to keep the same constraints and requirements that are currently placed on argc/argv.

What about Windows?

On Windows, when a non-conforming entry point is used instead of main, such as `WinMain or wmain, argc/argv` may not be directly available to the implementation.
In this case, a couple implementation strategies are possible:

- Have _Get_argc/_Get_argv return 0/a pointer to an empty string respectively.
- Convert the program arguments from __wargc or GetCommandLineA. However, we should note that such a conversion is not free, and most importantly, may lead to text encoding errors (resulting in badly encoded arguments) as it involves a conversion from UTF-16 (the encoding of __wargc items) to the local code page, and such conversion may be lossy.

As such, in scenarios in which no standard main function is defined, it is perfectly reasonable to return 0 from _Get_argc. This mirrors the status quo on windows, for example __argv is defined only in a non-Unicode Win32 environment, and only __wargv is defined otherwise.

Did you consider exposing arc/argv as variables instead?

Yes. Functions seem slightly more flexible and familiar, there is existing experience for both approaches.

Do we really need `_Get_argc`?

Argv is null-terminated so `_Get_argc` is not strictly necessary, but it avoids traversing the array in scenarios where we only want to know the number of elements, and so it is somewhat important for performance. As an example, programs frequently check whether they've been given the correct number of arguments, and if not, display help text for how to use the application. This common scenario should not have to pay the cost of traversing the entire list of arguments to get the count.

Did you consider `const char* _Get_arg(int index)`?

This is another possibility, although it is less efficient, and not an existing practice. It would, however, mirror the getenv API more closely.

Did you consider a single function returning a struct?

```
struct __arguments {
    int argc;
    const char* const * argv;
};
__arguments _Get_args(void);
```

This is another possible API.

Wording

All wording is relative to WG14 N2731.

Add to 7.22.4 Communication with the environment

7.22.4.□ The `_Get_argc` function

Synopsis

```
#include <stdlib.h>
int _Get_argc(void);
```

Description: The **`_Get_argc`** function returns the number of program parameters available to the host environment as described in 5.1.2.2.1.

Returns: If the program defines a main function with parameters, the _Get_argc function returns the same value as is passed for the argc argument to the main function. If the program defines a main function with no parameters, the _Get_argc function returns the

number of arguments available to the host environment. In both cases, the value returned obeys the same constraints as described in 5.1.2.2.1.

7.22.4.﷿ The `_Get_argv` function

Synopsis

```
#include <stdlib.h>
const char * const *_Get_argv(void);
```

Description:

The `_Get_argv` function returns the arguments available to the host environment that are optionally passed to the main function at startup. The `_Get_argv` function need not avoid data races with other threads of execution that modify the argument list.[footenote)] The implementation shall behave as if no library function calls the _Get_argv function.

Note: The value returned by `_Get_argv` remains valid for the duration of the program's lifetime.

Returns

If the program defines a main function with parameters, the _Get_argv function returns the same pointer value as is passed for the argv argument to the main function. If the program defines a main function with no parameters, the _Get_argv function returns a pointer to the arguments available to the host environment. In both cases, the pointer returned obeys the same constraints as described in 5.1.2.2.1.

[footnote)] The argument list passed to main, if any, is mutable.

References

Thanks to JeanHeyd Meneide and Robert C. Seacord for their valuable feedback on this paper.

References

n2731 Working draft — October 18, 2021 -
http://open-std.org/JTC1/SC22/WG14/www/docs/n2731.pdf