jank is C++

Jul 11, 2025 · Jeaye Wilkerson

If you've wondered how much a solo dev can build for seamless C++ interop on a quarter, you're about to find out. In April, jank was unable to reach into C++ at all. Toward the end of this post, I'll show some real world examples of what works today. Before that, though, I want to help you look out for the sponsorship this quarter, *not only* by all of my individual Github sponsors, but also by Cloyunists Together. I also want to say thank you to Vassil Vassilev and Lang Hames for building the necessary tech in Clang and LLVM for jank to actually do all of this magic. Let's get into it!

Memory management

In the past month, I have implemented manual memory management via `cpp/new` and `cpp/delete` . This uses jank's GC allocator (currently bdwgc), rather than `malloc` , so using `cpp/delete` isn't generally needed. However, if `cpp/delete` is used then memory collection can be eager and more deterministic.

The implementation has full bdwgc support for destructors as well, so both manual deletion and automatic collection will trigger non-trivial destructors.

```
(let [l (cpp/int. 500)
      p (cpp/new cpp/int l)]
  (assert (= l (cpp/* p))))
```

True and false

To avoid any implicit jank object casting, we can now use `cpp/true` and `cpp/false` , which are straight up C++ bools. These come in handy when trying to keep the generated IR as lean as possible, compared to using `true` or `false` and having jank do automatic conversions from Clojure land into C++ land. Going forward, jank will add support for `#cpp` reader macros, as an easy way to get C++ literals, similar to `#js` in ClojureScript and `#dart` in ClojureDart.

Complex type strings

It's possible to represent a lot of possible types using normal Clojure syntax. This month, I also extended that to include pointer types within symbols. For example, `cpp/int*` will give you a C++ `int*` type. However, when spaces or commas are required, such as with templates, Clojure's symbols become too limiting. In those cases, we can now use `(cpp/type "std::map<std::string, int>*")` . This will evaluate to a type which can be used in type position for `cpp/cast` , `cpp/new` , and so on.

Complex type constructors

With the new complex type syntax, we run into a problem. Clojure uses a `.` suffix to convey a constructor call, but we don't want to include a `.` suffix in `cpp/type` strings since that's not valid C++ syntax. To remedy this, jank now treats the suffix on types to be optional. If you call a type, it's considered a constructor call. Shout out ClojureDart for doing this first.

```
(let [ll (cpp/long 5)
      l2 ((cpp/type "long") 5)]
  (assert (cpp/== ll l2)))
```

Opaque boxes

In the JVM, every class implicitly inherits from `Object` . This allows Clojure's data structures to just store `Object` and not need to worry about all of the possible types which could be used. In the native land, however, every type is standalone by default. Even if you have a base object type in your own code, none of your dependencies will use it. The only way to refer to any type would be a `void*` . When we do this, though, the type information about that data is lost. It's up to the developer to correctly add that type information back at a later time, by casting that `void*` back to some other pointer.

This month, I built a structure for this, called opaque boxes. The idea is that you can take any raw native pointer and box it into a jank object using `cpp/box` . From there, that object can be used with jank's data structures, passed around, compared (by pointer value), etc. When you want to pull it back out, there is a special `cpp/unbox` form to do so while specifying the type of the data. It's entirely up to you to do this correctly, as it is in C or C++. Here's what it looks like, combined with `cpp/new` , for some pretend C++ kv store. Note that we don't need to specify the type to `cpp/box` , since the compiler already knows the type.

```
(ns example.kv)

(def db (delay (cpp/box (cpp/new cpp/my.db))))

(defn get! [k]
  (let [db (cpp/unbox cpp/my.db* @db)]
    (cpp/.get db (str k))))
```

Pre-compiled headers (PCH)

Seamless C++ interop with the jank runtime requires Clang to JIT process jank's C++ headers. This is costly and can affect startup time, so I've set up precompilation of those headers. This needs to be done per-machine, so jank will do it after install, when you first run jank. As jank is updated, it will automatically recompile the PCH as needed.

Stability

A great deal of work has gone into finding ways to break jank's seamless interop. C++ is such a large language, I have hundreds of interop tests. In the past month, I've found various crashes relested to arrays, global pointers, static references, function pointers, variadics, C function calls, and both Clang and LLVM-related issues regarding PCHs, IR optimizations, etc. This is an ongoing pursuit, at the bleeding edge of interop tech and jank's test suite is how I build confidence that the system works well.

Static typing

As a side note, for anyone who hadn't considered this yet, every bit of jank's seamless interop is statically typed. It is C++. There is no runtime reflection, no guess work, and no hints. If the compiler can't find a member, or a function, or a particular overload, you will get a compiler error. I think that this is an interesting way to start thinking about jank, Clojure, and static types. It also paves the way to start expanding that type info into other parts of the jank program.

Interlude

Before I show off some practical examples of jank doing C++ things, please consider subscribing to jank's mailing list. This is going to be the best way to make sure you stay up to date with jank's releases, jank-related talks, workshops, and so on.

Practical examples

Now that seamless interop actually works, we can try to do some things we might normally do in C++. The hardest part of this is coming up with good examples which fit well in a blog post!

Hello world via streams

Basically every C++ developer starts by including `iostream` and using `std::cout` . Here, we don't need to worry about `operator <<` returning a `std::ostream&` and that not being convertible to a jank object, since it's in statement position. If we didn't have the `nil` at the end of the function, we'd get a compiler error, since jank would try to automatically box the `std::ostream&` and find that there's no way to do it.

```
(cpp/raw "#include <iostream>")

(defn -main [& args]
  (cpp/<< cpp/std.cout (cpp/cast cpp/std.string "Hello, world!")
  nil)
```

We don't yet have the ability to do `#cpp "Hello, world!\n"` , to get a raw C string, but that's coming soon. That alone will clean up a lot of common string use cases.

JSON pretty printer

Ok, cranking up the complexity a bit, let's bring in a third party library. JSON for Modern C++ is likely the most popular C++ JSON library there is. It's header-only, so if we just download the standalone header, we can JIT include it. To turn this into a full program, we can also reach into `std::ifstream` for file reading. This program will take a JSON file as an argument, parse it, and then output the pretty printed JSON to stdout.

```
(cpp/raw "#include <fstream>")
(cpp/raw "#include \"json.hpp\"")

(defn -main [& args]
  (let [file (cpp/std.ifstream. (cpp/cast cpp/std.string (fir
       json (cpp/nlohmann.json.parse file)]
    (println (cpp/.dump json 2))))
```

The thing I love most about this example is how we're weaving between Clojure and C++ on basically every line. Yet it all just works.

FTXUI

I'm going to crank things up one more notch, since things are getting fun. Here's a working example which uses FTXUI to lay out terminal output using flexbox. However, in the Clojure way, we provide a pure data hiccup interface and the implementation handles the rest. Here, we take advantage of opaque boxing, to move ftxui objects between jank functions, we build up `std::vector` objects, and we rely on a ton of Clojure goodies to clean it up. So cool!

```
(cpp/raw "#include <ftxui/dom/elements.hpp>")
(cpp/raw "#include <ftxui/screen/screen.hpp>")
(cpp/raw "#include <ftxui/screen/string.hpp>")

(declare vbox)
(declare hbox)

(defn hiccup->element [h]
  (case (first h)
    :text (cpp/box (->> (str (second h))
                        cpp/ftxui.text
                        (cpp/new cpp/ftxui.Element)))
    :filler (cpp/box (cpp/new cpp/ftxui.Element (cpp/ftxui.fi
    :vbox (apply vbox (rest h))
    :hbox (apply hbox (rest h))))

(defmacro defbox [name f]
  `(defn ~name [& hiccup#]
     (let [elements# (cpp/ftxui.Elements.)]
       (doseq [h# hiccup#]
         (let [e# (cpp/* (cpp/unbox cpp/ftxui.Element* (hicco
           (cpp/.push_back elements# e#)))
       (cpp/box (cpp/new cpp/ftxui.Element (~f elements#))))))
(defbox hbox cpp/ftxui.hbox)
(defbox vbox cpp/ftxui.vbox)

(defn render-hiccup [hiccup]
  (let [document (->> (hiccup->element hiccup)
                      (cpp/unbox cpp/ftxui.Element*)
                      cpp/*)
        screen (cpp/ftxui.Screen.Create (cpp/ftxui.Dimension.
                                        (cpp/ftxui.Dimension.
    (cpp/ftxui.Render screen document)
    (cpp/.Print screen)
    (println)))

(defn -main [& args]
  (render-hiccup [:vbox
                  [:hbox
                   [:text "north-west"]
                   [:filler]
                   [:text "north-east"]]
                  [:filler]
                  [:hbox
                   [:filler]
                   [:text "center"]
                   [:filler]]
                  [:filler]
                  [:hbox
                   [:text "south-west"]
                   [:filler]
                   [:text "south-east"]]]))
```

A quick note about Clasp

The OG in the C++ Lisp space is Clasp, created by Dr. Christian Schafmeister. Clasp also integrates with LLVM and it uses MPS for a GC. Though jank and Clasp differ greatly, both in that Clasp is Common Lisp and jank is Clojure as well in how they approach C++ interop, they are two bold attempts to bridge two otherwise distant languages. I reached out to Dr. Schafmeister several years ago, when I started on jank, and we discussed C++ and Lisps. His work has been a big inspiration for jank.

What's next?

Phew. I've accomplished a ton this quarter and I'm extremely pleased with what jank can do now. Still, the work on seamless interop isn't finished and more work will be needed on it before jank can be released. A big issue is that I didn't have time to tackle automatic destructor calls for stack allocated C++ objects. I have one of my mentees, Jianling, helping out on the work here, to ensure that we can get it done soon.

On top of that, the largest issues comes from Clang and LLVM directly. One of the indicators of jank's seamless interop being unprecedented is the bugs and missing features I'm finding in Clang and LLVM. These are generally quite slow to address, since I'm relying on the volunteer time of the experts in those areas to help me out. jank would not exist without them. There are still cases where some interop code can trigger a crash in Clang and we'll have to tackle them as they come up. However, one of the best ways to speed this along is more funding, so that I can pay these Clang and LLVM experts for their time. Please consider becoming a Github Sponsor ♡ to *make this more feasible.*

Looking forward to the new quarter, the main focus will be packaging and distribution. I want to make jank easy to build everywhere and even easier to install. On top of that, I need to address all of the small pain points, various bugs, lack of tooling, etc. After packaging and distribution is stabilized, the rest of the year will be spent on bug fixes, tooling, and documentation. After that, we'll have the alpha launch!

Would you like to help out?

1. Join the community on Slack
2. Join the design discussions or pick up a ticket on GitHub
3. Considering becoming a Sponsor ♡
4. **Better yet, reach out to discuss corporate sponsorship!**