openjdk /
jextract

Code Pull requests 4 Security Insights

https://openjdk.org/projects/code-tools

GPL-2.0 license
192 stars 37 forks 7 watching Activity
Public repository

master

Branches Tags

JornVernee Backport: 7903449: Jextract generates structs that cannot be compiled ... on Apr 14 87

View code

README.md

Jextract

`jextract` is a tool which mechanically generates Java bindings from a native library headers. This tools leverages the clang C API in order to parse the headers associated with a given native library, and the generated Java bindings build upon the Foreign Function & Memory API. The `jextract` tool was originally developed in the context of Project Panama (and then made available in the Project Panama Early Access binaries).

Getting jextract

Pre-built binaries for jextract are periodically released here. These binaries are built from the `master` branch of this repo, and target the foreign memory access and function API in the latest mainline JDK (for which binaries can be found here).

Alternatively, to build jextract from the latest sources (which include all the latest updates and fixes) please refer to the building section below.

Using jextract

To understand how `jextract` works, consider the following C header file:

```
//point.h
struct Point2d {
    double x;
    double y;
};

double distance(struct Point2d);
```

We can run `jextract`, as follows:

```
jextract --source -t org.jextract point.h
```

We can then use the generated code as follows:

```
import java.lang.foreign.*;
import static org.jextract.point_h.*;
import org.jextract.Point2d;

class TestPoint {
    public static void main(String[] args) {
        try (var arena = Arena.openConfined()) {
            MemorySegment point = arena.allocate(Point2d.$LAYOUT());
```

```
            Point2d.x$set(point, 3d);
            Point2d.y$set(point, 4d);
            distance(point);
        }
    }
}
```

As we can see, the `jextract` tool generated a `Point2d` class, modelling the C struct, and a `point_h` class which contains static native function wrappers, such as `distance`. If we look inside the generated code for `distance` we can find the following:

```
static final FunctionDescriptor distance$FUNC = FunctionDescriptor.of(Constants$root.C_DOUBLE$LAYOUT,
    MemoryLayout.structLayout(
         Constants$root.C_DOUBLE$LAYOUT.withName("x"),
         Constants$root.C_DOUBLE$LAYOUT.withName("y")
    ).withName("Point2d")
);
static final MethodHandle distance$MH = RuntimeHelper.downcallHandle(
    "distance",
    constants$0.distance$FUNC
);

public static MethodHandle distance$MH() {
    return RuntimeHelper.requireNonNull(constants$0.distance$MH,"distance");
}
public static double distance ( MemorySegment x0) {
    var mh$ = distance$MH();
    try {
        return (double)mh$.invokeExact(x0);
    } catch (Throwable ex$) {
        throw new AssertionError("should not reach here", ex$);
    }
}
```

In other words, the `jextract` tool has generated all the required supporting code (`MemoryLayout`, `MethodHandle` and `FunctionDescriptor`) that is needed to call the underlying `distance` native function. For more examples on how to use the `jextract` tool with real-world libraries, please refer to the samples folder (building/running particular sample may require specific third-party software installation).

Command line options

The `jextract` tool includes several customization options. Users can select in which package the generated code should be emitted, and what the name of the main extracted class should be. If no package is specified, classes are generated in the unnamed package. If no name is specified for the main header class, then the header class name is derived from the header file name. For example, if jextract is run on foo.h, then foo_h will be the name of the main header class.

A complete list of all the supported options is given below:

Option	Meaning
`-D --define-macro <macro>=<value>`	define to (or 1 if omitted)
`--header-class-name <name>`	name of the generated header class. If this option is not specified, then header class name is derived from the header file name. For example, class "foo_h" for header "foo.h".
`-t, --target-package <package>`	target package name for the generated classes. If this option is not specified, then unnamed package is used.
`-I, --include-dir <dir>`	append directory to the include search paths. Include search paths are searched in order. For example, if `-I foo -I bar` is specified, header files will be searched in "foo" first, then (if nothing is found) in "bar".
`-l, --library <name | path>`	specify a library by platform-independent name (e.g. "GL") or by absolute path ("/usr/lib/libGL.so") that will be loaded by the generated class.
`--output <path>`	specify where to place generated files
`--source`	generate java sources instead of classfiles

Option	Meaning
`--dump-includes <String>`	dump included symbols into specified file (see below)
`--include-[function,constant,struct,union,typedef,var] <String>`	Include a symbol of the given name and kind in the generated bindings (see below). When one of these options is specified, any symbol that is not matched by any specified filters is omitted from the generated bindings.
`--version`	print version information and exit

Additional clang options

Users can specify additional clang compiler options, by creating a file named `compile_flags.txt` in the current folder, as described here.

Filtering symbols

To allow for symbol filtering, `jextract` can generate a *dump* of all the symbols encountered in an header file; this dump can be manipulated, and then used as an argument file (using the `@argfile` syntax also available in other JDK tools) to e.g. generate bindings only for a *subset* of symbols seen by `jextract`. For instance, if we run `jextract` with as follows:

```
jextract --dump-includes includes.txt point.h
```

We obtain the following file (`includes.txt`):

```
#### Extracted from: point.h

--include-struct Point2d    # header: point.h
--include-function distance # header: point.h
```

This file can be passed back to `jextract`, as follows:

```
jextract -t org.jextract --source @includes.txt point.h
```

It is easy to see how this mechanism allows developers to look into the set of symbols seen by `jextract` while parsing, and then process the generated include file, so as to prevent code generation for otherwise unused symbols.

Building & Testing

`jextract` depends on the C libclang API. To build the jextract sources, the easiest option is to download LLVM binaries for your platform, which can be found here (a version >= 9 is required). Both the `jextract` tool and the bindings it generates depend heavily on the Foreign Function & Memory API, so a suitable jdk 20 distribution is also required.

> ▶ Building older jextract versions

`jextract` can be built using `gradle`, as follows (on Windows, `gradlew.bat` should be used instead).

(**Note**: Run the Gradle build with a Java version appropriate for the Gradle version. For example, Gradle 7.5.1 supports JDK 18. Please checkout the Gradle compatibility matrix for the appropate JDK version needed for builds)

```
$ sh ./gradlew -Pjdk20_home=<jdk20_home_dir> -Pllvm_home=<libclang_dir> clean verify
```

> ▶ Using a local installation of LLVM

After building, there should be a new `jextract` folder under `build`. To run the `jextract` tool, simply run the `jextract` command in the `bin` folder:

```
$ build/jextract/bin/jextract
Expected a header file
```

Testing

The repository also contains a comprehensive set of tests, written using the jtreg test framework, which can be run as follows (again, on Windows, `gradlew.bat` should be used instead):

```
$ sh ./gradlew -Pjdk20_home=<jdk20_home_dir> -Pllvm_home=<libclang_dir> -Pjtreg_home=<jtreg_home> jtr
```

Note: running `jtreg` task requires `cmake` to be available on the `PATH`.

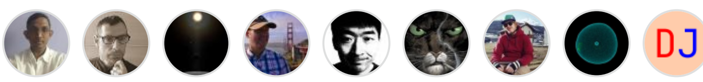
Releases

No releases published

Packages

No packages published

Contributors 9



Languages

- Java 85.9%
- C 12.3%
- Shell 1.4%
- Other 0.4%