How to Windows

Cross-platform installation and testing for Julia packages

Tony Kelman

@tkelman

JuliaCon
June 25, 2015

How to Support Windows

Cross-platform installation and testing for Julia packages

Tony Kelman

`@tkelman`

JuliaCon

June 25, 2015

Reasons to care about Windows support

- Education
  - Not all students can afford a Mac
  - Not everyone knows how to use Linux
- Some industries
  - Require proprietary Windows-only software
  - Conservative IT policies
  - Instrumentation and embedded hardware
- Masochism
  - Fun to find and fix the bugs no one else wants to touch

Reasons to care about Windows support

- Education
  - Not all students can afford a Mac
  - Not everyone knows how to use Linux
- Some industries
  - Require proprietary Windows-only software
  - Conservative IT policies
  - Instrumentation and embedded hardware
- Masochism
  - Fun to find and fix the bugs no one else wants to touch

Good news and bad news

- Biggest question for package authors - binary dependencies
- Is your package code pure Julia?
  - Base Julia is portable enough that most code will work
  - Few things to watch out for, shelling out, filesystem differences
  - Run your tests on platforms you don't use
- Are you wrapping a `C`, `C++`, or Fortran library?
  - Harder problem, but not hopeless
  - Automated tools for building and distributing binaries

Continuous integration for Julia packages on Windows

AppVeyor

- AppVeyor `http://www.appveyor.com` - free for open source projects
- Same idea as Travis CI `https://travis-ci.org`, but for Windows
- Add `appveyor.yml` to your repository and click one button to enable
  - Starts up a Windows VM on every commit and pull request
  - Installs your package, runs your tests, reports status back to GitHub
- Template `appveyor.yml` file in Example.jl package `https://github.com/JuliaLang/Example.jl/blob/master/appveyor.yml`

Example appveyor.yml file

```
environment:
  matrix:
  - JULIAVERSION: "julialang/bin/winnt/x86/0.3/julia-0.3-latest-win32.exe"
  - JULIAVERSION: "julialang/bin/winnt/x64/0.3/julia-0.3-latest-win64.exe"
  - JULIAVERSION: "julianightlies/bin/winnt/x86/julia-latest-win32.exe"
  - JULIAVERSION: "julianightlies/bin/winnt/x64/julia-latest-win64.exe"

install:
# Download most recent Julia Windows binary
  - ps: (new-object net.webclient).DownloadFile(
        $("http://s3.amazonaws.com/"+$env:JULIAVERSION),
        "C:\projects\julia-binary.exe")
# Run installer silently, output to C:\projects\julia
  - C:\projects\julia-binary.exe /S /D=C:\projects\julia

build_script:
# Need to convert from shallow to complete for Pkg.clone to work
  - IF EXIST .git\shallow (git fetch --unshallow)
  - C:\projects\julia\bin\julia -e "versioninfo();
      Pkg.clone(pwd(), \"Example\"); Pkg.build(\"Example\")"

test_script:
  - C:\projects\julia\bin\julia --check-bounds=yes -e "Pkg.test(\"Example\")"
```

Binary dependencies, the hard part

- What makes building scientific software on Windows hard?
- Windows is not POSIX - no `sh`, no coreutils
- Win32 system API, `NtQueryInformationFile` and other unspeakable horrors
- No built-in package management for native dependency handling
- Let's talk about compilers
  - Visual Studio, the platform native choice
    - World-class debugger and IDE
    - `C` is an afterthought, very recently started supporting C99 (mostly)
    - No Fortran compiler, no 64-bit inline assembly
  - Intel's compilers, the expensive option
    - High performance `C`, `C++`, and Fortran compilers
    - Has features missing from MSVC, but not free to install

Binary dependencies, the hard part

- What makes building scientific software on Windows hard?
- Windows is not POSIX - no `sh`, no coreutils
- Win32 system API, `NtQueryInformationFile` and other unspeakable horrors
- No built-in package management for native dependency handling
- Let's talk about compilers
  - Visual Studio, the platform native choice
    - World-class debugger and IDE
    - `C` is an afterthought, very recently started supporting C99 (mostly)
    - No Fortran compiler, no 64-bit inline assembly
  - Intel's compilers, the expensive option
    - High performance `C`, `C++`, and Fortran compilers
    - Has features missing from MSVC, but not free to install

Compilers and build systems, cont'd

- Open-source compilers
  - Clang, the newcomer
    - ⋆ Windows `C++` exception handling a work in progress
    - ⋆ No Fortran compiler
  - GCC (MinGW-w64), yes it does run everywhere
    - ⋆ `gfortran` only option for open source scientific community
    - ⋆ Excellent cross compilation support
- Let's talk about build systems
  - Autotools `./configure` assumes a POSIX shell
  - Virtually all Makefiles written for GNU `make`
  - CMake an emerging standard, not universally used yet
- Best way to build scientific libraries for Windows today?
  - Build with GCC using a POSIX environment like Cygwin, or MSYS2, or cross compile from Linux
  - Distribute just the binaries, Julia packages only need `dll` files
    - ⋆ Users do not need compilers or build environment at install or run time

Compilers and build systems, cont'd

- Open-source compilers
  - Clang, the newcomer
    - Windows `C++` exception handling a work in progress
    - No Fortran compiler
  - GCC (MinGW-w64), yes it does run everywhere
    - `gfortran` only option for open source scientific community
    - Excellent cross compilation support
- Let's talk about build systems
  - Autotools `./configure` assumes a POSIX shell
  - Virtually all Makefiles written for GNU `make`
  - CMake an emerging standard, not universally used yet
- Best way to build scientific libraries for Windows today?
  - Build with GCC using a POSIX environment like Cygwin, or MSYS2, or cross compile from Linux
  - Distribute just the binaries, Julia packages only need `dll` files
    - Users do not need compilers or build environment at install or run time

Building and providing Windows binaries

- Small, self-contained research code?
  - Manually compile and upload a `dll`
  - From Linux you can use `x86_64-w64-mingw32-gcc` to cross compile
  - Julia package just downloads compiled `dll` with `Binaries` provider from BinDeps.jl `https://github.com/JuliaLang/BinDeps.jl`
  - Add the following to a file `deps/build.jl` in your package

```
using BinDeps

@BinDeps.setup

libfoo = library_dependency("libfoo")

# Sources and BuildProcess providers for source build on Linux
# Homebrew.jl provider for binaries on OS X

provides(Binaries, URI(string("https://github.com/foo/libfoo/",
    "releases/download/v#.#.#/libfoo-#.#.#.tar.gz")),
    [libfoo], os = :Windows)

@BinDeps.install Dict([(:libfoo, :libfoo)])
```

openSUSE build service for cross compiling

- Established library with complicated dependencies?
  - openSUSE has many cross-compiled Windows packages `https://build.opensuse.org/project/show/windows:mingw:win64`
  - Easy to use automated build service, a little like GitHub
  - Upload source tarball, write a spec file with compilation instructions

```
%define _basename zeromq
Name:           mingw64-%{_basename}
Version:        4.0.5
Release:        0
#!BuildIgnore: post-build-checks
Summary:        Lightweight messaging kernel
License:        LGPL-3.0+
Group:          Productivity/Networking/Web/Servers
Url:            http://www.zeromq.org/
Source:         http://download.zeromq.org/%{_basename}-%{version}.tar.gz
BuildRequires:  mingw64-cross-binutils
BuildRequires:  mingw64-cross-gcc
BuildRequires:  mingw64-cross-gcc-c++
BuildRequires:  mingw64-cross-pkg-config
BuildRequires:  mingw64-filesystem
BuildRoot:      %{_tmppath}/%{name}-%{version}-build
%_mingw64_package_header_debug
BuildArch:      noarch
```

spec file for cross compiling

```
%package devel
Summary:        Development files for ZeroMQ
Group:          Development/Languages/C and C++
Requires:       %name = %version
Provides:       mingw64-libzmq-devel = %{version}

%_mingw64_debug_package

%prep
%setup -q -n %{_basename}-%{version}

%build
%{_mingw64_configure}                              ⟵ ./configure --host=x86_64-w64-mingw32
%{_mingw64_make} %{?_smp_mflags} V=1               ⟵ make

%install
%{_mingw64_make} DESTDIR=%{buildroot} install      ⟵ make install

%files
%defattr(-,root,root,-)
%doc COPYING COPYING.LESSER
%{_mingw64_bindir}/libzmq.dll
%{_mingw64_bindir}/curve_keygen.exe

%files devel
%defattr(-,root,root,-)
%doc AUTHORS ChangeLog COPYING COPYING.LESSER NEWS
%{_mingw64_includedir}/zmq*
%{_mingw64_libdir}/libzmq.dll.a
%{_mingw64_libdir}/pkgconfig/libzmq.pc

%changelog
```

WinRPM.jl for library installation

- WinRPM.jl `https://github.com/JuliaLang/WinRPM.jl` primarily written by `@vtjnash` and `@ihnorton`
- Parses RPM metadata, dependencies and latest versions, from openSUSE build service and downloads binaries
- Add the following to `deps/build.jl` in your package

```
using BinDeps

@BinDeps.setup

libfoo = library_dependency("libfoo")

# other providers for Linux, OS X

@windows_only begin
    using WinRPM
    provides(WinRPM.RPM, "foo", [libfoo], os = :Windows)
    # replace "foo" with "Name:" value from opensuse package
    # but without the "mingw64-" prefix
end

@BinDeps.install Dict([(:libfoo, :libfoo)])
```

Not an impossible problem

- A simple user experience can be achieved
- Packages can work on Windows even if developers don't use it
- Provide binaries, turn on automated testing

Questions?