- **IMPORTANT**: Before upgrading to a new version, **always check for breaking changes.**
- 👀 ➡️ Table of contents (ToC) is available by expanding the *Pages* sidebar to the right ➡️

Quick start

1. Install build prerequisites on your system
2. `git clone https://github.com/neovim/neovim`
3. `cd neovim && make`
   - If you want the **stable release**, also run `git checkout stable`.
   - If you want to install to a custom location, set `CMAKE_INSTALL_PREFIX`. See also Installing Neovim.
   - On BSD, use `gmake` instead of `make`.
   - To build on Windows, see the Building on Windows section.
4. `sudo make install`
   - Default install location is `/usr/local`

Notes:

- From the repository's root directory, running `make` will download and build all the needed dependencies and put the `nvim` executable in `build/bin`.
- Third-party dependencies (libuv, LuaJIT, etc.) are downloaded automatically to `.deps/`. See the FAQ if you have issues.
- After building, you can run the `nvim` executable without installing it by running `VIMRUNTIME=runtime ./build/bin/nvim`.
- If you plan to develop Neovim, install Ninja for faster builds. It will automatically be used.

Running tests

See test/README.md.

Building

First make sure you installed the build prerequisites. Now that you have the dependencies, you can try other build targets explained below.

The *build type* determines the level of used compiler optimizations and debug information:

- `Release`: Full compiler optimizations and no debug information. Expect the best performance from this build type. Often used by package maintainers.
- `Debug`: Full debug information; few optimizations. Use this for development to get meaningful output from debuggers like GDB or LLDB. This is the default if `CMAKE_BUILD_TYPE` is not specified.
- `RelWithDebInfo` ("Release With Debug Info"): Enables many optimizations and adds enough debug info so that when Neovim ever crashes, you can still get a backtrace.

So, for a release build, just use:

```
make CMAKE_BUILD_TYPE=Release
```

(Do not add a `-j` flag if `ninja` is installed! The build will be in parallel automatically.)

Afterwards, the `nvim` executable can be found in `build/bin`. To verify the build type after compilation, run:

```
./build/bin/nvim --version | grep ^Build
```

To install the executable to a certain location, use:

```
make CMAKE_INSTALL_PREFIX=$HOME/local/nvim install
```

CMake, our main build system, caches a lot of things in `build/CMakeCache.txt`. If you ever want to change `CMAKE_BUILD_TYPE` or `CMAKE_INSTALL_PREFIX`, run `rm -rf build` first. This is also required when rebuilding after a Git commit adds or removes files (including from `runtime`) — when in doubt, run `make distclean` (which is basically a shortcut for `rm -rf build .deps`).

By default (`USE_BUNDLED=1`), Neovim downloads and statically links its needed dependencies. In order to be able to use a debugger on these libraries, you might want to compile them with debug information as well:

```
make distclean
VERBOSE=1 DEBUG=1 make deps
```

Building on Windows

Windows / Cygwin

Install all dependencies the normal way, then build Neovim the normal way for a random CMake application (i.e. do not use the `Makefile` that automatically downloads and builds "bundled" dependencies).

The `cygport` repo contains Cygport files (e.g. `APKBUILD`, `PKGBUILD`) for all the dependencies not available in the Cygwin distribution, and describes any special commands or arguments needed to build. The Cygport definitions also try to describe the required dependencies for each one. Unless custom commands are provided, Cygport just calls `autogen` / `cmake`, `make`, `make install`, etc. in a clean and consistent way.

https://github.com/cascent/neovim-cygwin was built on Cygwin 2.9.0. Newer `libuv` should require slightly less patching. Some SSP stuff changed in Cygwin 2.10.0, so that might change things too when building Neovim.

Windows / MSYS2 / MinGW

From the MSYS2 shell, install these packages:

```
pacman -S \
    mingw-w64-x86_64-{gcc,libtool,cmake,make,perl,python2,pkg-config,ninja,diffutils}
\
    gperf
```

Now, from the Windows Command Prompt (`cmd.exe`), set up the `PATH` and build.

```
set PATH=c:\msys64\mingw64\bin;c:\msys64\usr\bin;%PATH%
```

Build using the `Ninja` generator:

```
mkdir .deps
cd .deps
cmake -G Ninja ..\third-party\
ninja
cd ..

mkdir build
cd build
cmake -G Ninja -D CMAKE_BUILD_TYPE=RelWithDebInfo ..
ninja
ninja install
```

If you cannot install neovim with `ninja install`, the following command will produce a zip archive, which you can extract anywhere. Then you can add the `bin/` directory inside it to your PATH variable and invoke `nvim` from anywhere in your command line.

```
cpack -G ZIP -C RelWithDebInfo
```

For 32-bit builds, adjust the package names and paths accordingly.

Windows / MSVC

Note: No one has already confirmed that building with the following steps is possible today. See build.ps1 for the confirmed steps.

1. Install Visual Studio (2017 or later) with the *Desktop development with C++* workload.
    - On 32-bit Windows, you will need this workaround.
2. Open the Neovim project. Visual Studio automatically starts the build.
3. **IMPORTANT**: Select `x86-Release` configuration instead of `x64-{Debug,Release}`.
    - You can build with the `x64-Release` configuration if `cmake -G "Visual Studio 15 2017 Win64"` is used to build the dependencies. But the *Debug* configurations will not work because certain dependencies need to be linked with the release version of the C runtime.
    - If the build fails, it may be because Visual Studio started the build with `x64-{Debug,Release}` before you switched the configuration to `x86-Release`.
        - Right-click *CMakeLists.txt* → *Delete Cache*.
        - Right-click *CMakeLists.txt* → *Generate Cache*.

Note: If you want to build from the command line (i.e. invoke the `cmake` commands yourself), make sure you have the Visual Studio environment variables properly set -- with the Visual Studio Developer Command Prompt, or `Import-VisualStudioVars` from this PowerShell module. This is to make sure that `luarocks` finds the Visual Studio installation, and doesn't fall back to MinGW with errors like `'mingw32-gcc' is not recognized as an internal or external command`.

Windows / CLion

1. Install CLion.
2. Open the Neovim project in CLion.
3. Select *Build* → *Build All in* '*Release*'.

Localization

Localization build

A normal build will create `.mo` files in `build/src/nvim/po`.

- If you see `msgfmt: command not found`, you need to install [`gettext`]. On most systems, the package is just called `gettext`.

Localization check

To check the translations for `$LANG`, run `make -C build check-po-$LANG`. Examples:

```
make -C build check-po-de
make -C build check-po-pt_BR
```

- Use `ninja` instead of `make` if applicable.
- `check-po-$LANG` generates a detailed report in `./build/src/nvim/po/check-${LANG}.log`. (The report is generated by `nvim`, not by `msgfmt`.)

Localization update

To update the `src/nvim/po/$LANG.po` file with the latest strings, run the following:

```
make -C build update-po-$LANG
```

- Replace `make` with `ninja` if applicable.
- **Note**: Run `src/nvim/po/cleanup.vim` after updating.

Compiler options

To see the chain of includes, use the `-H` option (#918):

```
echo '#include "./src/nvim/buffer.h"' | \
> clang -I.deps/usr/include -Isrc -std=c99 -P -E -H - 2>&1 >/dev/null | \
> grep -v /usr/
```

- `grep -v /usr/` is used to filter out system header files.
- `-save-temps` can be added as well to see expanded macros or commented assembly.

Xcode and MSVC project files

CMake has a `-G` option for exporting to multiple project file formats, such as Xcode and Visual Studio.

For example, to use Xcode's static analysis GUI (#167), you need to generate an Xcode project file from the Neovim Makefile (where `neovim/` is the top-level Neovim source code directory containing the main `Makefile`):

```
cmake -G Xcode neovim
```

The resulting project file can then be opened in Xcode.

Custom Makefile

You can customize the build process locally by creating a `local.mk`, which is referenced at the top of the main `Makefile`. It's listed in `.gitignore`, so it can be used across branches. **A new target in `local.mk` overrides the default make-target.**

Here's a sample `local.mk` which adds a target to force a rebuild but *does not* override the default-target:

```
all:

rebuild:
    rm -rf build
    make
```

Third-party dependencies

Reference the Debian package (or alternatively, the Homebrew formula) for the precise list of dependencies/versions.

To build the bundled dependencies using CMake:

```
mkdir .deps
cd .deps
cmake ../third-party
make
```

By default the libraries and headers are placed in `.deps/usr`. Now you can build Neovim:

```
mkdir build
cd build
cmake ..
make
```

How to build without "bundled" dependencies

1. Manually install the dependencies:
    - gperf libuv libluv libtermkey libvterm luajit lua-lpeg lua-mpack msgpack-c tree-sitter unibilium
2. Do the "CMake dance": create a `build` directory, switch to it, and run CMake:

    ```
    mkdir build
    cd build
    cmake ..
    ```

    If all the dependencies are not available in the package, you can use only some of the bundled dependencies as follows (example of using `ninja`):

    ```
    mkdir .deps
    cd .deps
    cmake ../third-party/ -DUSE_BUNDLED=OFF -DUSE_BUNDLED_LIBVTERM=ON -
    DUSE_BUNDLED_TS=ON
    ninja
    cd ..
    mkdir build
    cd build
    cmake ..
    ```
3. Run `make`, `ninja`, or whatever build tool you told CMake to generate.

    - Using `ninja` is strongly recommended.

Debian 10 (Buster) example:

```
sudo apt install gperf luajit luajit-5.1-dev lua-mpack lua-lpeg libunibilium-dev
libmsgpack-dev libtermkey-dev
mkdir .deps
cd .deps
cmake ../third-party/ -DUSE_BUNDLED=OFF -DUSE_BUNDLED_LIBUV=ON -DUSE_BUNDLED_LUV=ON
-DUSE_BUNDLED_LIBVTERM=ON -DUSE_BUNDLED_TS=ON
ninja
cd ..
mkdir build
cd build
cmake ..
ninja
```

Example of using a Makefile

- Example of using a package with all dependencies:

  ```
  make USE_BUNDLED=OFF
  ```

- Example of using a package with some dependencies:

  ```
  make BUNDLED_CMAKE_FLAG="-DUSE_BUNDLED=OFF -DUSE_BUNDLED_LUV=ON -
  DUSE_BUNDLED_TS=ON -DUSE_BUNDLED_LIBVTERM=ON -DUSE_BUNDLED_LIBUV=ON"
  ```

Build prerequisites

General requirements (see #1469):

- Clang or GCC version 4.4+
- CMake version 3.10+, built with TLS/SSL support
    - Optional: Get the latest CMake from an installer or the Python package (`pip install cmake`)

Platform-specific requirements are listed below.

Ubuntu / Debian

```
sudo apt-get install ninja-build gettext libtool libtool-bin autoconf automake cmake
g++ pkg-config unzip curl doxygen
```

Note: `libtool-bin` is only required for Ubuntu 16.04 / Debian 8 and newer.

CentOS / RHEL / Fedora

If you're using CentOS/RHEL 6, you need at least `autoconf` version 2.69 for compiling the `libuv` dependency. See also https://github.com/joyent/libuv/issues/1158.

```
sudo yum -y install ninja-build libtool autoconf automake cmake gcc gcc-c++ make
pkgconfig unzip patch gettext curl
```

openSUSE

```
sudo zypper install ninja libtool autoconf automake cmake gcc-c++ gettext-tools curl
```

Arch Linux

```
sudo pacman -S base-devel cmake unzip ninja tree-sitter curl
```

Alpine Linux

```
apk add build-base cmake automake autoconf libtool pkgconf coreutils curl unzip
gettext-tiny-dev
```

Void Linux

```
xbps-install base-devel cmake curl git
```

NixOS / Nix

Starting from NixOS 18.03, the Neovim binary resides in the `neovim-unwrapped` Nix package (the `neovim` package being just a wrapper to setup runtime options like Ruby/Python support):

```
cd path/to/neovim/src
```

Drop into `nix-shell` to pull in the Neovim dependencies:

```
nix-shell '<nixpkgs>' -A neovim-unwrapped
```

Configure and build:

```
rm -rf build && cmakeConfigurePhase
buildPhase
```

Tests are not available by default, because of some unfixed failures. You can enable them via adding this package in your overlay:

```
  neovim-dev = (super.pkgs.neovim-unwrapped.override  {
    doCheck=true;
  }).overrideAttrs(oa:{
    cmakeBuildType="debug";

    nativeBuildInputs = oa.nativeBuildInputs ++ [ self.pkgs.valgrind ];
    shellHook = ''
      export NVIM_PYTHON_LOG_LEVEL=DEBUG
      export NVIM_LOG_FILE=/tmp/log
      export VALGRIND_LOG="$PWD/valgrind.log"
    '';
  });
```

and replacing `neovim-unwrapped` with `neovim-dev`:

```
nix-shell '<nixpkgs>' -A neovim-dev
```

Starting November 2020, Neovim contains a Nix flake in the `contrib` folder, with 3 packages:

- `neovim` to run the nightly
- `neovim-debug` to run the package with debug symbols
- `neovim-developer` to get all the tools to develop on `neovim`

Thus you can run Neovim nightly with `nix run github:neovim/neovim?dir=contrib`. Similarly to develop on Neovim: `nix develop github:neovim/neovim?dir=contrib#neovim-developer`.

FreeBSD

```
sudo pkg install cmake gmake libtool sha automake pkgconf unzip wget gettext curl
```

If you get an error regarding a `sha256sum` mismatch, where the actual SHA-256 hash is `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`, then this is your issue (that's the `sha256sum` of an empty file). Also, make sure Wget is installed. LuaRocks has bad interactions with cURL, at least under FreeBSD, and will die with a PANIC in LuaJIT when trying to install a rock.

OpenBSD

```
doas pkg_add gmake cmake libtool unzip autoconf-2.69p2 automake-1.15p0 curl
export AUTOCONF_VERSION=2.69
export AUTOMAKE_VERSION=1.15
```

For older versions of OpenBSD less than 6.1, the `autoconf-2.69` and `automake-1.15` ports may have different `p` suffixes.

Build can sometimes fail when using the top level `Makefile`, apparently due to some third-party component (see #2445-comment). The following instructions use CMake:

```
mkdir .deps
cd .deps
cmake ../third-party/
gmake
cd ..
mkdir build
cd build
cmake ..
gmake
```

macOS

macOS / Homebrew

1. Install Xcode Command Line Tools: `xcode-select --install`
2. Install Homebrew
3. Install Neovim build dependencies:

```
brew install ninja libtool automake cmake pkg-config gettext curl
```

- **Note**: If you see Wget certificate errors (for older macOS versions less than 10.10):

```
brew install curl-ca-bundle
echo CA_CERTIFICATE=$(brew --prefix curl-ca-bundle)/share/ca-bundle.crt >>
~/.wgetrc
```

- **Note**: If you see `'stdio.h' file not found`, try the following:

```
open
/Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg
```

macOS / MacPorts

1. Install Xcode Command Line Tools: `xcode-select --install`
2. Install MacPorts
3. Install Neovim build dependencies:

```
sudo port install ninja libtool autoconf automake cmake pkgconfig gettext
```

- **Note**: If you see Wget certificate errors (for older macOS versions less than 10.10):

```
sudo port install curl-ca-bundle
echo CA_CERTIFICATE=/opt/local/share/curl/curl-ca-bundle.crt >> ~/.wgetrc
```

- **Note**: If you see `'stdio.h' file not found`, try the following:

```
open
/Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg
```

Building for older macOS versions

From a newer macOS version, to build for older macOS versions, you will have to set the macOS deployment target:

```
make CMAKE_BUILD_TYPE=Release MACOSX_DEPLOYMENT_TARGET=10.13 DEPS_CMAKE_FLAGS="-
DCMAKE_CXX_COMPILER=$(xcrun -find c++)"
```

Note that the C++ compiler is explicitly set so that it can be found when the deployment target is set.