Useful tools for developers

David Grellscheid

MCnet

Tools for every stage of a project

Project planning

Developer communication

Project dependencies

Source code management

Editing

Building

Testing

Debugging

Profiling

User interaction

Project planning

Easy collaborative editing

Record of decisions

To-do list

Needs to be exportable!

https://trello.com/

https://docs.google.com/

Developer communication

Email lists
(archive functionality!)

Chat clients (export?)
Slack, Mattermost

https://www.discourse.org/

Wiki/Ticket system
(e.g. Trac, see later)

Build environment, dependencies

Make use of existing package managers

Linux: apt / yum

OS X: MacPorts, Homebrew

Multi-system: Vagrant

Spend time looking for existing libraries

Boost, GNU scientific library, …

Source code management

Good version control is vital

Choice is either git or hg

Many GUI choices

'hg serve' as minimal option

Repository hosting:
github, bitbucket, …

Merge tool: meld

Editor

emacs, vim, sublime, atom

Crucial features:

auto-indentation, syntax highlighting
intelligent search (declaration - definition)
one-command build

Compiler

gcc / clang

Compile with both

Understand the options (-Wall !)
do not blindly apply flags

Build tools

learn ‘make’, not just for code!

Collects sequence of steps reproducibly
One-action rebuild

Build tools

Standard package build on Linux

./configure; make; make install

For published, portable projects,
do not write Makefiles yourself.

Use autoconf / automake / libtool
to ensure familiar behaviour (CC CXX etc.)

(Or use cmake if you hate your users)

Debugging / profiling

Learn at least basics of gdb / lldb

valgrind for memory leaks

OS X: Instruments app

linux-tools has ‘perf’, very useful

https://perf.wiki.kernel.org/index.php/Tutorial

User interaction

Bug reporting: Trac tickets

Documentation: sphinx, Doxygen

MCnet

Testing

Testing can be automatic.
Debugging needs humans

It must be automatic,
otherwise it won't happen

"Looks right" is a dangerous criterion,
used far too often

Unit Testing

Especially important in interpreted languages

Every function of program tested with inputs

Possible approach: Test-driven development.
only write code when there's a failing test!

Python: nosetest, doctest

C++: Boost.Test

(also interesting: Quickcheck-based approaches)

Integration Testing

Does the program as a whole work?

Does it work for all target systems?

Most common tools for
continuous integration testing:

https://jenkins.io/index.html
http://buildbot.net/

Automatic detection of repository changes

Anything else?