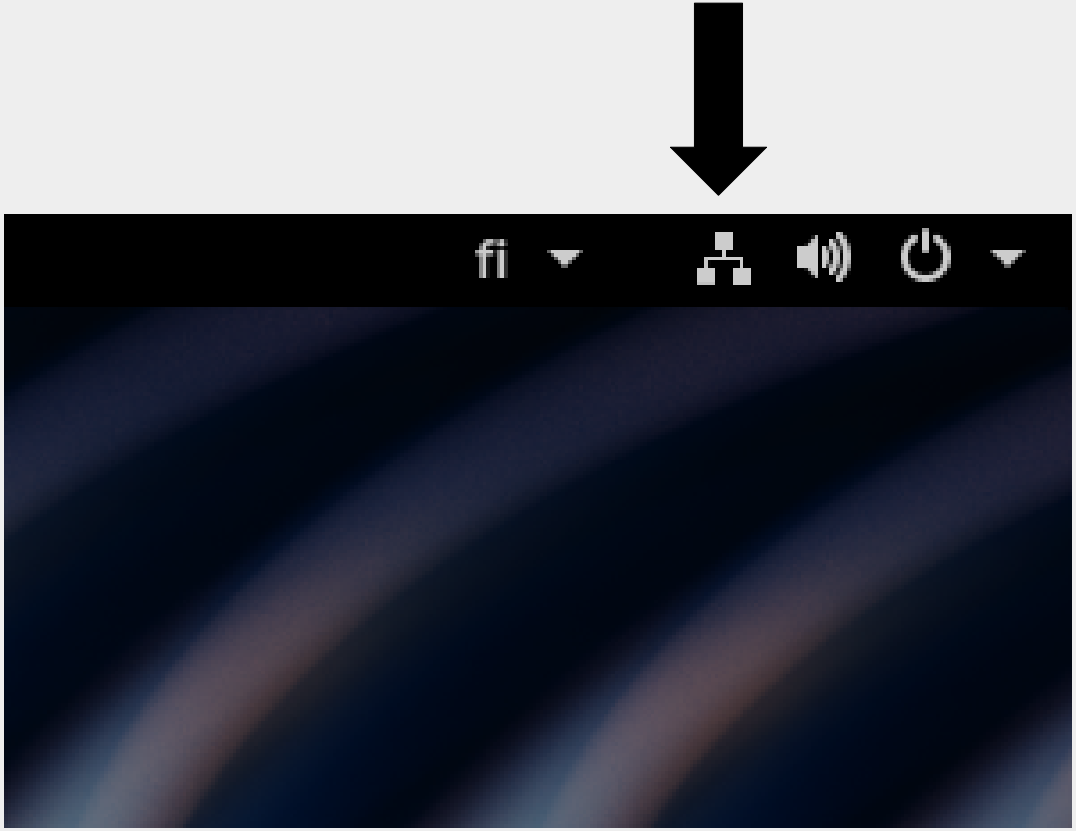
Behind the Scenes of a C++ Build System

Jussi Pakkanen
@jpakkane

Project lead
Meson Build System
https://meson.build

This talk is not a monolith

- Rather a set of independent micro~~services~~talks
- together they shall illuminate optimization, design, dependency management and multiplatform issues
- Writing this presentation surprised me, hopefully it will do the same to you


fi

The three edged sword

- Users: *the applet shows hotspots that should not exist, it breaks my network*
- Network devs: *if you are showing networks the user does not want to see, it's your problem, wontfix*
- Managers: *your refusal to fix the issue will look bad on your quarterly review*

Messenger shooting

Whoever makes a problem visible
gets blamed for its existance.

Build systems make *many* problems visible.

An illustrating example

Process invocation.

The simplest thing in the world.

Basic approach

- Store command and arguments as a string array
- Transfer data between programs in format that has native array support such as JSON
- Eventually call `execve()`
- Done, what was so hard about that?

Windows does not provide a process invocation function that takes arrays.

That actually works.

Every tool that does cross platform process invocation must treat commands as strings.

Even though it is the
wrong abstraction.

And thus you get to play …

The Quoting Game

- Quote the string in the build definition file
- Cmd.exe quote the array
- Ninja quote the cmd quote
- Shell quote the array
- Ninja quote the shell quote
- Quote the response file
- Ninja quote the quoted response file

There is no standard for response file quoting.

Some tools take their arguments cmd quoted on the command line but shell quoted in rsp files.

Some tools have buggy rsp file parsers.

“But surely no-one would …”

```
gstenumtypes.h: $(gst_headers)
	$(AM_V_GEN)$(GLIB_MKENUMS) \
	--fhead "#ifndef __GST_ENUM_TYPES_H__\n#define __GST_ENUM_TYPES_H__\n\n#include <glib-object.
	--fprod "\n/* enumerations from \"@filename@\" */\n" \
	--vhead "GST_API GType @enum_name@_get_type (void);\n#define GST_TYPE_@ENUMSHORT@ (@enum_name
	--ftail "G_END_DECLS\n\n#endif /* __GST_ENUM_TYPES_H__ */" \
	$^ > gstenumtypes.h
```

In defense of Microsoft

Every platform and toolset has similar issues.

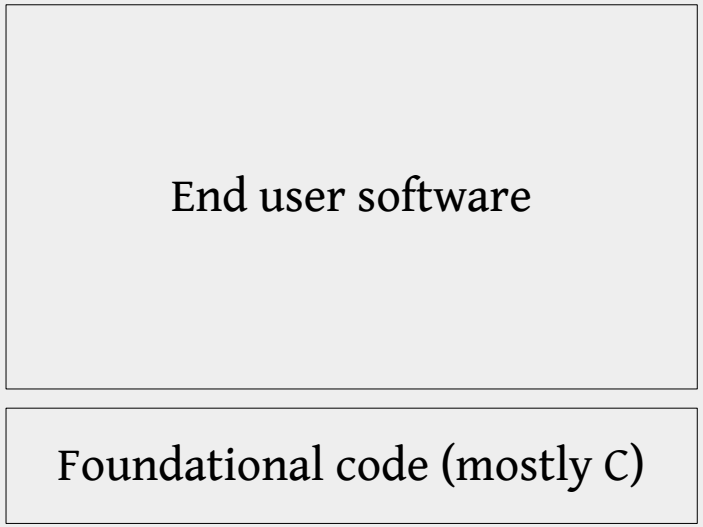
Language X has an awesome build system Y, why doesn't C++ have one?

Many a redditor

C++ (and especially C) are special

Things in the lowest layers of the stack are weird.

Two types of build systems



"App" build systems

"Core" build systems

Bootstrapping new platforms

- Cross compile minimal set of tools: shell, compiler, libc/libstdc++, minimal build tools
- Become self hosting as soon as possible
- Compile the rest natively
- Important for philosophical and legal reasons
- Most recently happened with RISC-V

Foundational code may not use a build system that requires Java, Haskell or even C++17.

Even Python was stretching it.

Comment from a BSD developer

We can't use Meson in (this)BSD until you reimplement it in either C or Perl.

Optimizing build speed

Shared libraries are awesome!

Even if you eventually ship your app
as a single statically linked blob.

Not only is shared linking faster, it can make linking steps *disappear completely*.

A shared library is fully specified
by the list of symbols it exports.

The algorithm (from Chromium)

- Build entire project
- Extract each library's exported symbols to a file
- When library gets rebuilt, compare new symbol list
- If lists are the same, relinking can be skipped
- In practice a bit more complicated but not by much

Implementation-only changes
go from *O(n)* to *O(1)*.

On design

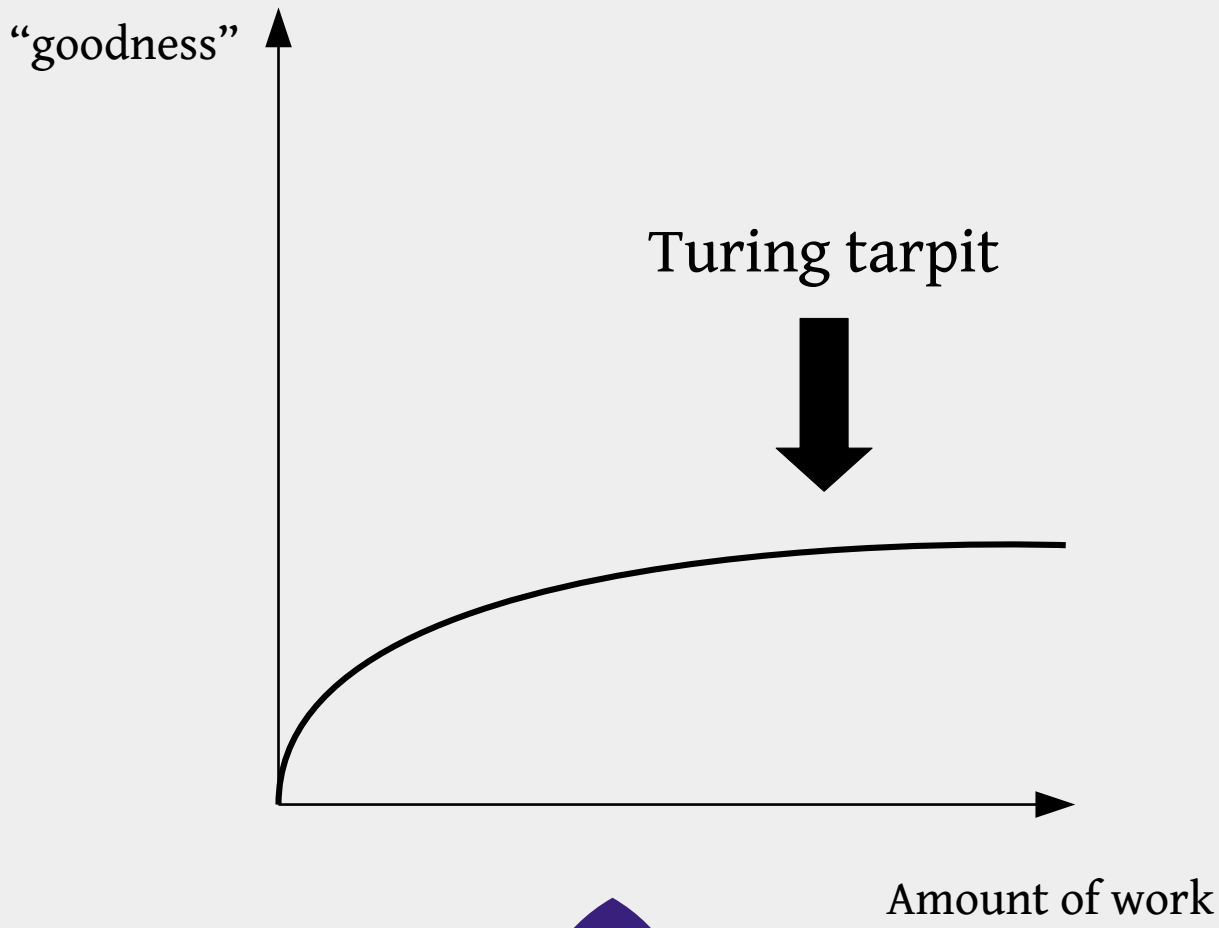
I must have a full Turing complete language because my builds are unique.

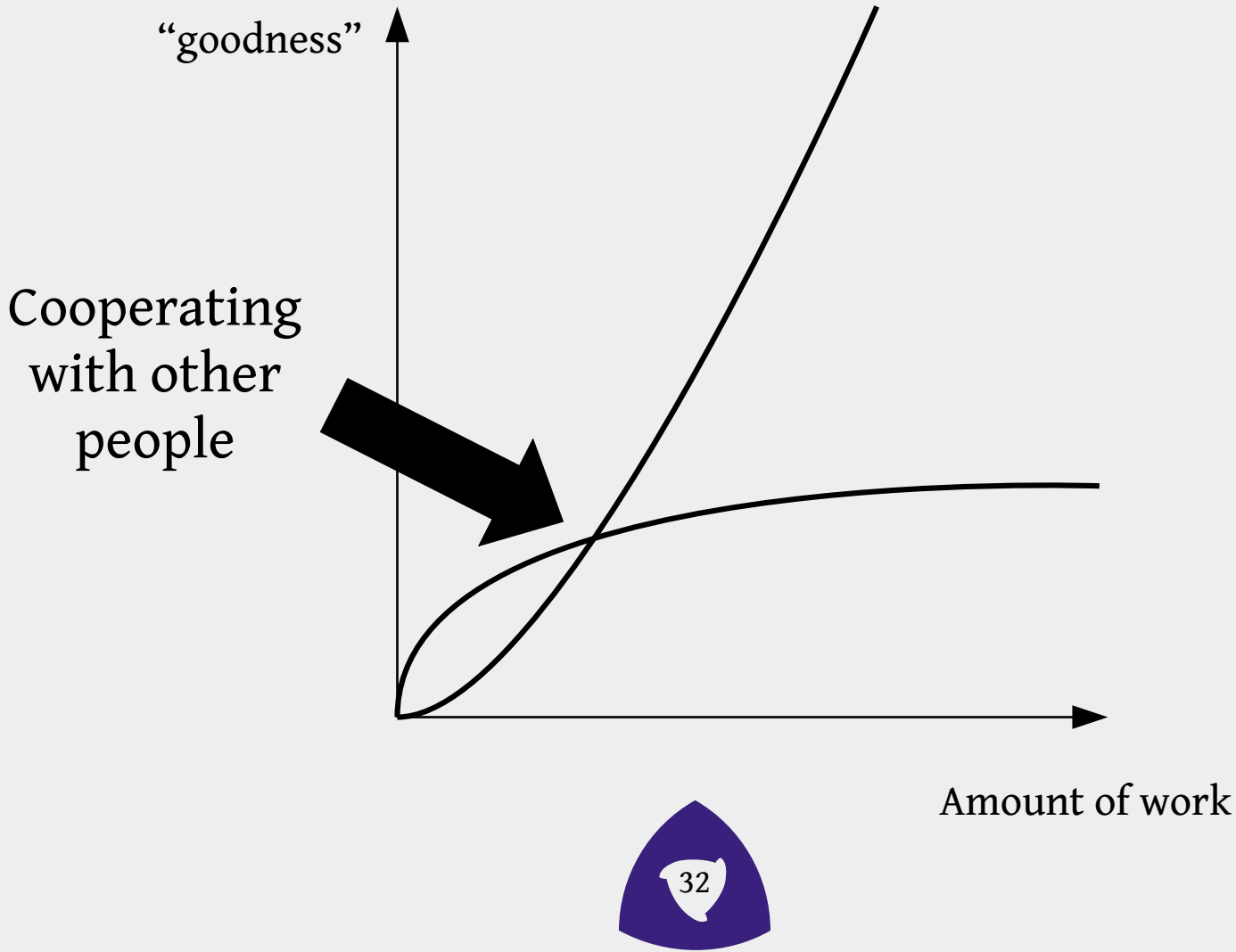
Jussi's Law of Programmers

The problem with programmers is that if you give them the chance, they will start programming.

Corollary

There is no limit to the amount of work a programmer is willing to do to avoid reading documentation.


“goodness”
Turing tarpit
Amount of work


“goodness”
Cooperating with other people
Amount of work

You can’t be “just a little bit” pregnant.

You can’t be “just a little bit” Turing complete.

Dependency management

Making it smooth and scalable

What has worked thus far?

1. Install everything to a common prefix, provide dependency info via Pkg-Config or similar
2. Build everything with a single build system in a single build dir in a single go

Nothing else has been proven to work.

In particular, using two different build systems in a single build dir **does not work**. No matter how much people want it to.

What problems do these approaches have?

For approach #1

Multiple versions of dependencies in the same filesystem get mixed up unexpectedly.

For approach #2

The entire world needs to standardise on a single build system.

Ergo

Everything will be awful forever.

Unless …

The problem with approach #1 were multiple versions of the same dependency on the same filesystem.

So what if they weren't
on the same file system?

Is there a mechanism to isolate filesystem parts from each other?

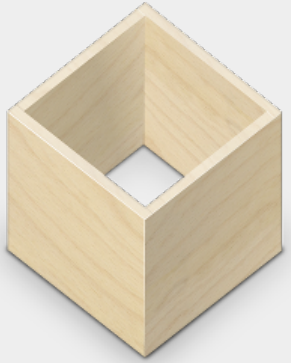
Filesystem namespaces

Isn’t this just Docker?

The future is already here,
it's just not very evenly distributed.

William Gibson

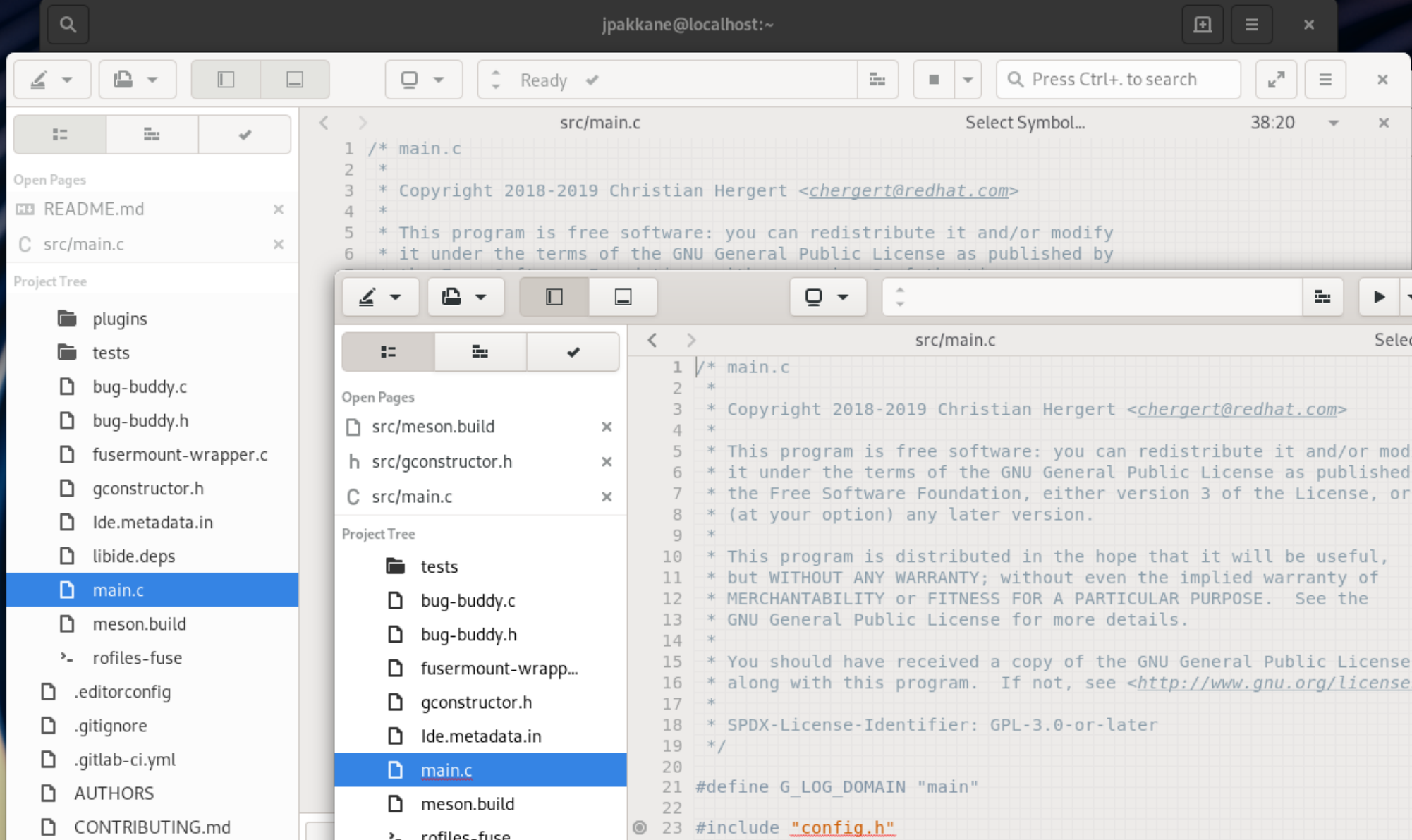
Flatpak



Library files come from custom root dir.
Work files come from user's home directory.

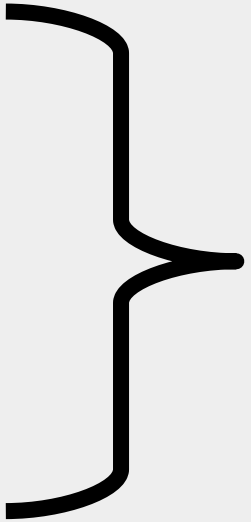
GNOME Builder

World's first "container native" IDE.


jpakkane@localhost:~
Ready
Press Ctrl+. to search
src/main.c
Select Symbol...
38:20
Open Pages
README.md
src/main.c
Project Tree
plugins
tests
bug-buddy.c
bug-buddy.h
fusermount-wrapper.c
gconstructor.h
lde.metadata.in
libide.deps
main.c
meson.build
rofiles-fuse
.editorconfig
.gitignore
.gitlab-ci.yml
AUTHORS
CONTRIBUTING.md
/* main.c
*
* Copyright 2018-2019 Christian Hergert <chergert@redhat.com>
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
src/main.c
Select Sym
Open Pages
src/meson.build
src/gconstructor.h
src/main.c
Project Tree
tests
bug-buddy.c
bug-buddy.h
fusermount-wrapp...
gconstructor.h
lde.metadata.in
main.c
meson.build
rofiles-fuse
/* main.c
*
* Copyright 2018-2019 Christian Hergert <chergert@redhat.com>
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
* SPDX-License-Identifier: GPL-3.0-or-later
*/
#define G_LOG_DOMAIN "main"
#include "config.h"

Flatpak manifest file (imitation)

```
Runtime: freedesktop 18/08
Dependencies:
  SDL-ttf-2 2.0.11
    <sha checksum here>
  other-dependency
    <sha checksum here>
  third-dependency
    <sha checksum here>
  etc
```



deterministic
cacheable

Assuming a reliable distributed file system,
build parallelization becomes trivial.

Bringing it all together

Why have build systems been deemed so terrible?

They surface problems in underlying systems
They have to solve problems other languages do not have
They can not use functionality other languages do
They have not been subject to rigorous optimization
Short term thinking drives them to Turing tarpits

And most importantly …

Because we have been trying to implement a kernel space construct entirely in user space.