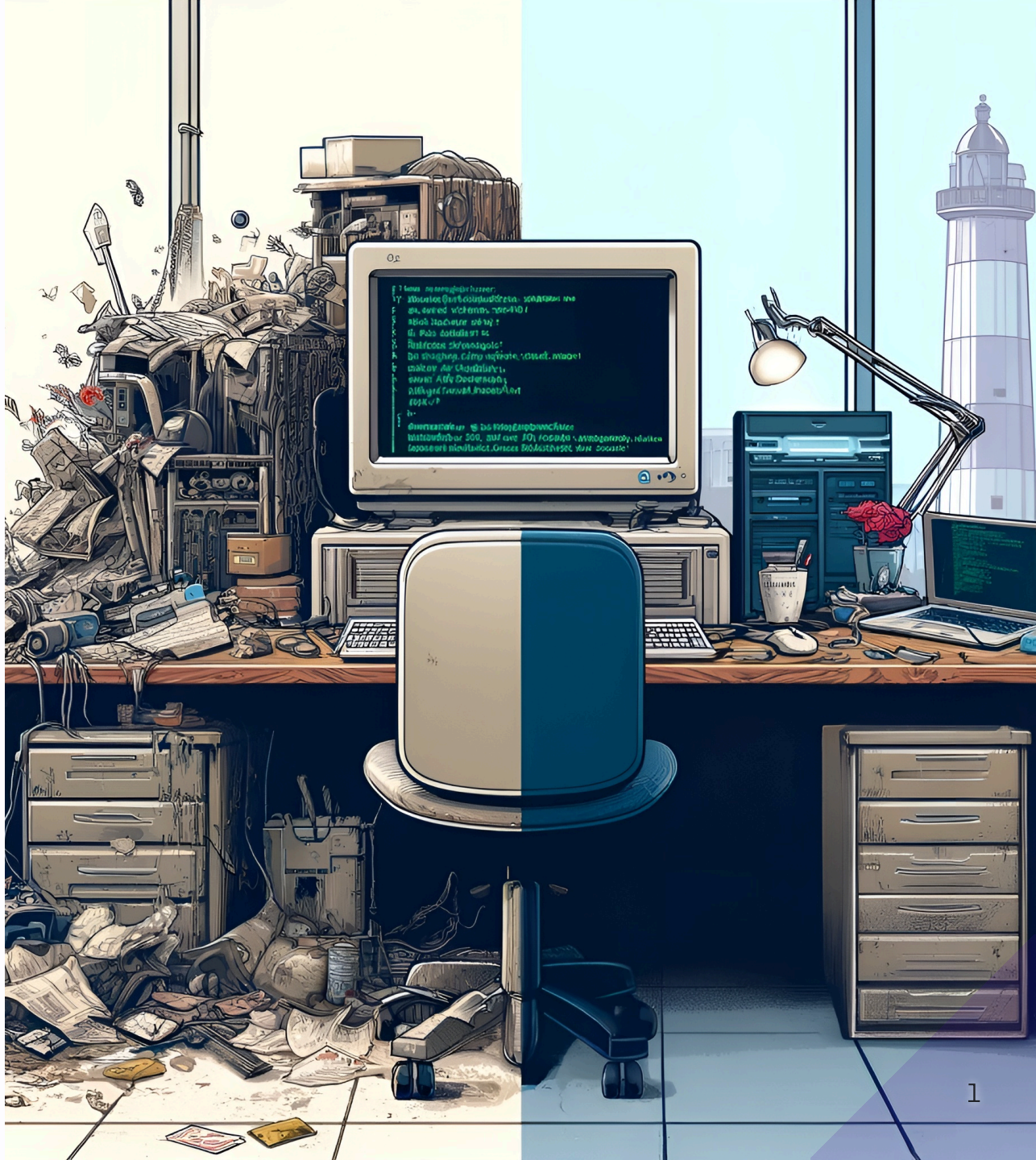
Customising clang-tidy to modernise your legacy C++ code

Mike Crowe

Meeting C++

15th November 2024

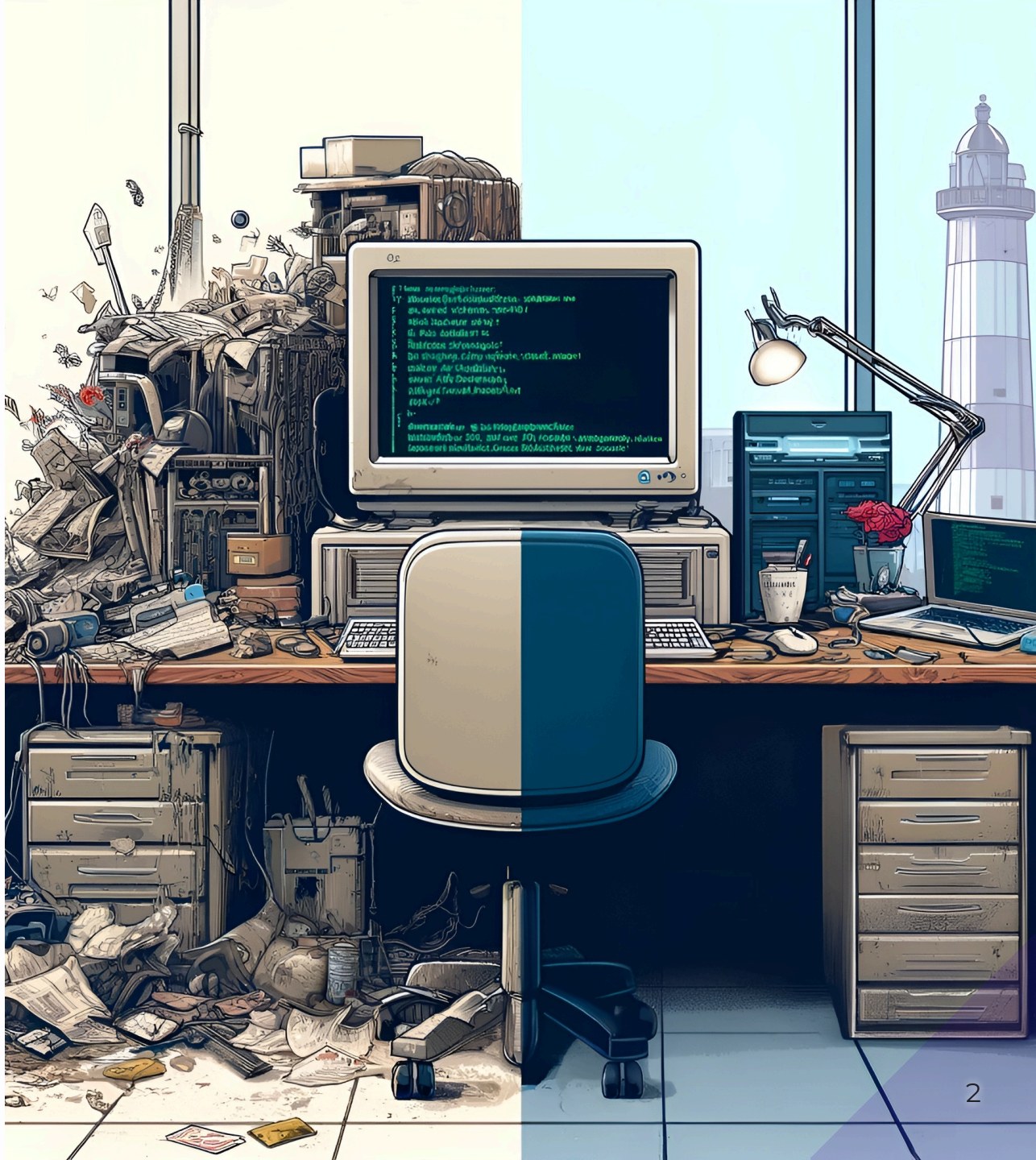


Customising clang-tidy to modernise your legacy C++ code

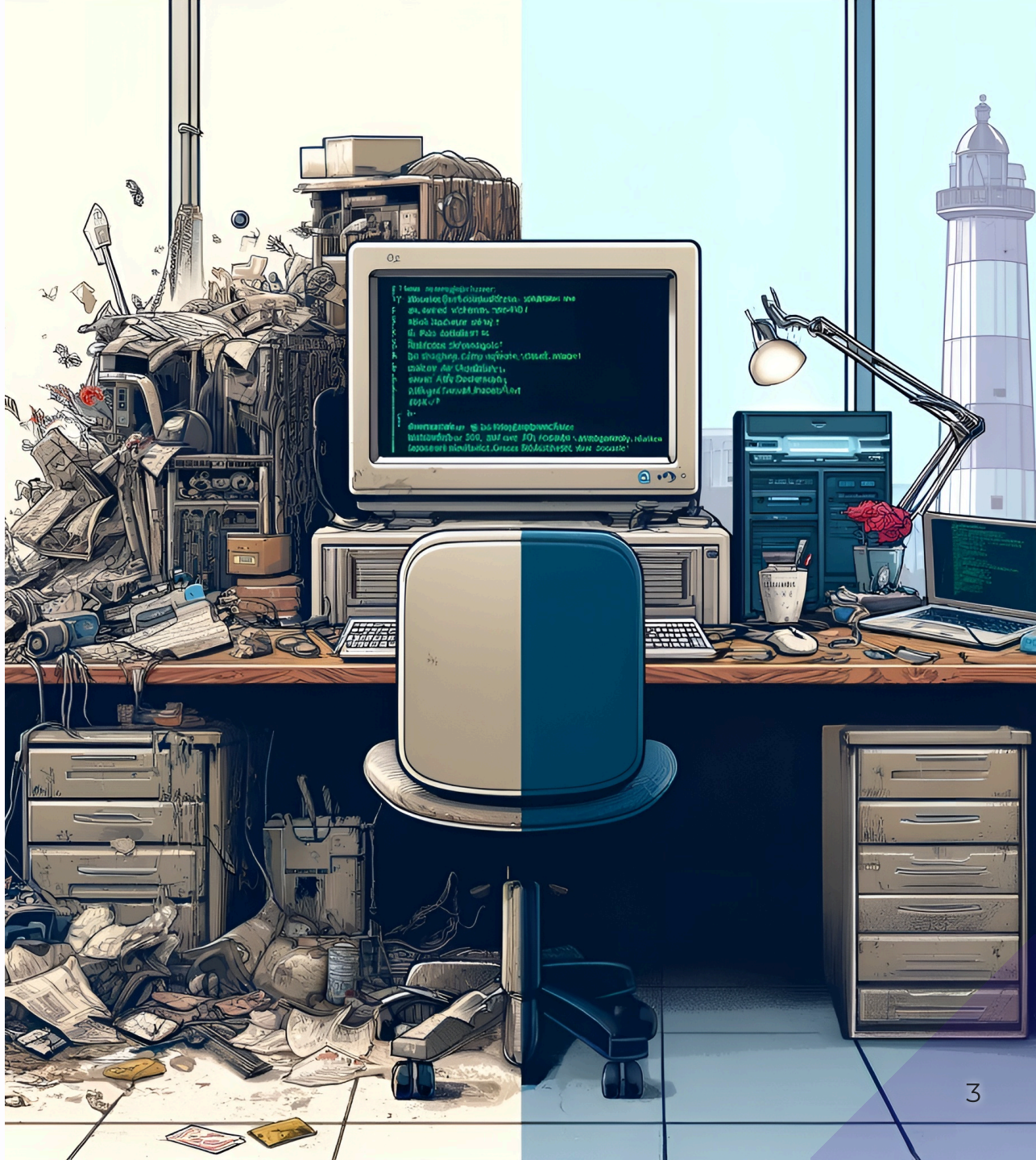
Mike Crowe

Meeting C++

15th November 2024



Or, how you too can change ~7,500 lines automatically and safely



About Me

- C++
- Embedded Linux
- Media players
- mac@mcrowe.com
- mastodon.social/@mikecrowe

About BrightSign OS

- Embedded Linux using OpenEmbedded/Yocto
- Main C++ application started in 2007 but with older parts
- Built on lots of C and C++ libraries

C++98-era Logging

```
static DebugLog LOG;
static ErrorLog LOG_ERROR;

int main(int argc, char *argv[]) {
  LOG("Argument count: %d\n", argc);
  if (argc != 1)
        LOG_ERROR("Too many arguments\n");
}
```

C++98-era Logging Implementation 1

```
#if DEBUG>0
typedef AlwaysLog DebugLog;
#else
typedef NeverLog DebugLog;
#endif
typedef AlwaysLog ErrorLog;
```

C++98-era Logging Implementation 2

```
#if DEBUG>0
typedef AlwaysLog DebugLog;
#else
typedef NeverLog DebugLog;
#endif
typedef AlwaysLog ErrorLog;

// For the logging disabled case we do nothing, and the compiler
// can throw everything away.
class NeverLog
{
public:
  void operator()(const char *, ...) { /* Nothing */ }
};
```

C++98-era Logging Implementation 3

```
#if DEBUG>0
typedef AlwaysLog DebugLog;
#else
typedef NeverLog DebugLog;
#endif
typedef AlwaysLog ErrorLog;

// For the logging disabled case we do nothing, and the compiler
// can throw everything away.
class NeverLog
{
public:
  void operator()(const char *, ...) { /* Nothing */ }
};

// For the logging enabled case we need to actually do something.
class AlwaysLog
{
public:
  void operator()(const char *, ...);
};
```

Implementation of `AlwaysLog`

```
#include <stdarg.h>

void AlwaysLog::operator()(const char *format, ...) {
  va_list va;
  va_start(va, format);
  vfprintf(stderr, format, va);
  va_end(va);
}
```

C++98-era Logging Again

```
static DebugLog LOG;
static ErrorLog LOG_ERROR;

int main(int argc, char *argv[]) {
  LOG("Argument count: %d\n", argc);
  if (argc != 1)
        LOG_ERROR("Too many arguments\n");
}
```

All was fine for about a decade

until...

32-bit Linux targets

- `long` = 32 bits
- `int64_t` = `long long`

```
void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%d b=%lld\n", i, l, a, b);
}

f(42, 85L, 16, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
```

64-bit Linux targets 1

- `long` = 64 bits
- `int64_t` = `long`

```
void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%d b=%ld\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
```

64-bit Linux targets 2

- `long` = 64 bits
- `int64_t` = `long`

```
void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%d b=%lld\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

If you try to use `%lld` on 64-bit Linux:

```
warning: format '%lld' expects argument of type 'long long int',
         but argument 2 has type 'long int'
```

32 & 64-bit Windows targets

- `long` = 32 bits
- `int64_t` = `long long`

```
void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%d b=%lld\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
```

<cinttypes> macros

```
#include <cinttypes>

void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%" PRId32 " b=%" PRId64 "\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
```

<cinttypes> macros

```
// #define PRIi32 "d"
// #define PRIi64 "ld" // 64-bit Linux: long
//  or
// #define PRIi64 "lld" // 32-bit Linux: long long
//  or
// #define PRIi64 "I64d" // Windows: __int64

void f(int i, long l, int32_t a, int64_t b) {
  LOG("i=%d l=%ld a=%" PRId32 " b=%" PRId64 "\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
```

The Solution: `fmt::fprintf`

- https://fmt.dev/

```
void f(int i, long l, int32_t a, int64_t b) {
  // {fmt} doesn't care about the exact integer type
  fmt::fprintf(stderr, "i=%d l=%ld a=%d b=%ld\n", i, l, a, b);
  fmt::fprintf(stderr, "i=%d l=%ld a=%d b=%lld\n", i, l, a, b);
  fmt::fprintf(stderr, "i=%d l=%d a=%d b=%d\n", i, l, a, b);
}

f(42, 85L, 16U, 1024);
```

outputs

```
i=42 l=85 a=16 b=1024
i=42 l=85 a=16 b=1024
i=42 l=85 a=16 b=1024
```

AlwaysLog using fmt::fprintf

```
template <typename... Args>
void AlwaysLog::operator()(const char *format, Args&&... args) {
    try {
        fmt::fprintf(stderr, format, std::forward<Args>(args)...);
    } catch (const std::exception &e) {
        fmt::fprintf(stderr, "Log message '%s' threw '%s'\n", format, e.what());
    }
}
```

More details in ACCU Overload #145

Logging `std::string`

- `{fmt}` is just as happy to be passed
  - `std::string` or
  - `const char *`

```
void f(const char *s1, const std::string &s2) {
  // call to c_str() is now unnecessary and ugly
  LOG("s1=%s, s2=%s\n", s1, s2.c_str());

  // this works just as well
  LOG("s1=%s, s2=%s\n", s1, s2);
}

f("Hello", "world");
```

outputs

```
s1=Hello s2=World
s1=Hello s2=World
```

clang-tidy

- https://clang.llvm.org/extra/clang-tidy/
- Part of Clang/LLVM
- A set of automatic checks for suboptimal C++ code, many with automatic fixes
- Don't need to build with Clang

Running clang-tidy: simple

```
clang-tidy --checks='-*,modernize-loop-convert' -fix loop.cpp
```

```
std::string concat(const std::vector<std::string> &vs) {
  std::string result;
  for (auto i = vs.begin(); i != vs.end() ; ++i)
    result += *i;
  return result;
}
```

to

```
std::string concat(const std::vector<std::string> &vs) {
  std::string result;
  for (const auto & v : vs)
    result += v;
  return result;
}
```

Running clang-tidy: `.clang-tidy`

`.clang-tidy` file:

```
Checks: -*,modernize-loop-convert
CheckOptions: { MinConfidence: risky }
```

Running clang-tidy: `compile_commands.json`

```
[ {
    "directory": "/home/mac/src/oe2/sources/bs/",
    "file": "apps/brightsign/main.cpp",
    "output": ".../apps/brightsign/cobra-debug1/main.o",
    "arguments": [ ".../recipe-sysroot-native/usr/bin/aarch64-oe-linux/aarch64-oe-linux-g++",
        "-march=armv8-a+crc+crypto", "-fstack-protector-strong", "-D_FORTIFY_SOURCE=2",
        "-Wformat", "-Wformat-security", "-Werror=format-security",
        "--sysroot=.../recipe-sysroot",
        ...
    ]
} ]
```

- Generated by CMake, Meson, ...
- Bear "Build Ear" https://github.com/rizsotto/Bear
- Automatically used if found

Running clang-tidy: Fixing multiple files

- One file at a time is slow.
- Fixing header files is racy.
- Use `run-clang-tidy.py` wrapper.

Running clang-tidy: Build variants

- clang-tidy only fixes code seen by the compiler
- Beware of conditionally-compiled code
- Beware of comments

Anatomy of a clang-tidy Check

- Class that inherits from `ClangTidyCheck`
- Override `registerMatchers` to add required AST matchers
- Override `check` to react to each match found during parsing
- Other overrides for customising other functionality
- Part of the llvm-project source code

Implementing `registerMatchers`

- Add matchers written as expressions using the Clang AST matcher DSL
- Built up into specific matchers required by the check
- Matchers form a DSL but are valid C++ code

redundant-string-cstr matcher 1

```
const auto StringCStrCallExpr =
  /*...*/;




const auto AlwaysLogClassExpr =
  /*...*/;


const auto NeverLogClassExpr = /*...*/;

Finder->addMatcher(
  traverse(TK_AsIs,
           cxxOperatorCallExpr(hasOverloadedOperatorName("()"),
                               hasArgument(0, anyOf(AlwaysLogClassExpr, NeverLogClassExpr),
                               forEachArgumentWithParam(StringCStrCallExpr,
                                                        parmVarDecl())))),
  this);
```

redundant-string-cstr matcher 2

```
const auto StringCStrCallExpr =
  /*...*/;



const auto AlwaysLogClassExpr =
  expr(hasType(hasCanonicalType(hasDeclaration(cxxRecordDecl(
                    isSameOrDerivedFrom("::AlwaysLog"))))));

const auto NeverLogClassExpr = /* ... isSameOrDerivedFrom("::NeverLog") */;

Finder->addMatcher(
  traverse(TK_AsIs,
           cxxOperatorCallExpr(hasOverloadedOperatorName("()"),
                               hasArgument(0, anyOf(AlwaysLogClassExpr, NeverLogClassExpr),
                               forEachArgumentWithParam(StringCStrCallExpr,
                                                        parmVarDecl())))),
  this);
```

redundant-string-cstr matcher 3

```
const auto StringCStrCallExpr =
    cxxMemberCallExpr(on(StringExpr.bind("arg")),
                      callee(memberExpr().bind("member")),
                      callee(cxxMethodDecl(hasAnyName("c_str", "data"))))
        .bind("call");

const auto AlwaysLogClassExpr =
  expr(hasType(hasCanonicalType(hasDeclaration(cxxRecordDecl(
                      isSameOrDerivedFrom("::AlwaysLog"))))));

const auto NeverLogClassExpr = /* ... isSameOrDerivedFrom("::NeverLog") */;

Finder->addMatcher(
  traverse(TK_AsIs,
           cxxOperatorCallExpr(hasOverloadedOperatorName("()"),
                               hasArgument(0, anyOf(AlwaysLogClassExpr, NeverLogClassExpr),
                               forEachArgumentWithParam(StringCStrCallExpr,
                                                        parmVarDecl()))),
  this);
```

Implementing `check`

- Extract information from the AST.
- Emit a "diagnostic" with the code changes required.

redundant-string-cstr fix 1

```
const auto *Call = Result.Nodes.getNodeAs<CallExpr>("call");
const auto *Arg = Result.Nodes.getNodeAs<Expr>("arg");
const auto *Member = Result.Nodes.getNodeAs<MemberExpr>("member");
```

redundant-string-cstr fix 2

```
const auto *Call = Result.Nodes.getNodeAs<CallExpr>("call");
const auto *Arg = Result.Nodes.getNodeAs<Expr>("arg");
const auto *Member = Result.Nodes.getNodeAs<MemberExpr>("member");

std::string ArgText =
  Member->IsArrow() ? utils::fixit::formatDereference(*Arg, *Result.Context)
                    : tooling::fixit::getText(*Arg, *Result.Context).str();

// The first line turns pstr->c_str() into *pstr
// The second line turns str.c_str() into str
```

redundant-string-cstr fix 3

```
const auto *Call = Result.Nodes.getNodeAs<CallExpr>("call");
const auto *Arg = Result.Nodes.getNodeAs<Expr>("arg");
const auto *Member = Result.Nodes.getNodeAs<MemberExpr>("member");

std::string ArgText =
  Member->IsArrow() ? utils::fixit::formatDereference(*Arg, *Result.Context)
                    : tooling::fixit::getText(*Arg, *Result.Context).str();

diag(Call->getBeginLoc(), "redundant call to %0")
      << Member->getMemberDecl()
      << FixItHint::CreateReplacement(Call->getSourceRange(), ArgText);
```

redundant-string-cstr result

Running the check on:

```
void f(const std::string &str) {
    LOG("Hello %s", str.c_str());
}
```

results in:

```
<source>:3:21: warning: redundant call to 'c_str' [readability-redundant-string-cstr]
    3 |     LOG("Hello %s", str.c_str());
      |                     ^~~~~~~~~~
      |                     str
```

"Lit" tests

```
void test_log(const std::string &s1, const std::string &s2)
{
  LOG("One:%s Two:%s\n", s1.c_str(), s2.c_str());
  // CHECK-MESSAGES: [[@LINE-1]]:26: warning: redundant call to 'c_str' [readability-redundant-string-cstr]
  // CHECK-MESSAGES: [[@LINE-2]]:38: warning: redundant call to 'c_str' [readability-redundant-string-cstr]
  // CHECK-FIXES: LOG("One:%s Two:%s\n", s1, s2);
}
```

1,000 lines changed

```
-             LOG("Looking for normal update in '%s'\n", f->c_str());
+             LOG("Looking for normal update in '%s'\n", *f);
```

6,000 to go

Are the types in the `printf` format string necessary?

- `fmt::fprintf` knows the types - why do we need to say `%d`, `%u` and `%s`?
- `fmt::println` with new Python-like format string language now standardised as `std::println`

```
fmt::println("{}: {}", "Count", 42);
```

A `printf` to `std::print` converter

```
printf("Hello %s\n", "world");
```

becomes

```
std::println("Hello {}", "world");
```

- Clang already knows how to parse `printf` format strings so it can check them.
- How hard can it be? It's just replacing `%something` with `{}`, isn't it?

Printing chars

```c
char c = 'C';
// both passed as int
printf("%c %d\n", c, c);
```

needs to become

```rust
char c = 'C';
std::println("{} {:d}", c, c);
```

Output: `C 67`

Printing chars

```c
char c = 'C';
// still both passed as int
printf("%c %hhd\n", c, c);
```

needs to become

```rust
char c = 'C';
std::println("{} {:d}", c, c);
```

Output: `C 67`

Sign conversion during printing 1

```
unsigned int ui = 2'147'483'648;
int i = -2'147'483'648;
printf("d: %d %d\nu: %u %u\n", i, ui, i, ui);
```

Output:

```
d: -2147483648 -2147483648
u: 2147483648 2147483648
```

Sign conversion during printing 2

```cpp
unsigned int ui = 2'147'483'648;
int i = -2'147'483'648;
printf("d: %d %d\nu: %u %u\n", i, ui, i, ui);
std::print("d: {} {}\nu: {} {}\n", i, ui, i, ui);
```

`printf` Output:

```
d: -2147483648 -2147483648
u: 2147483648 2147483648
```

`std::print` Output:

```
d: -2147483648 2147483648
u: -2147483648 2147483648
```

Sign conversion during printing

```
unsigned int ui = 2'147'483'648;
int i = -2'147'483'648;
printf("d: %d %d\nu: %u %u\n", i, ui, i, ui);
```

needs to become:

```
unsigned int ui = 2'147'483'648;
int i = 2'147'483'648;
std::println("d: {} {}\nu: {} {}",
          i, static_cast<int>(ui),
          static_cast<unsigned int>(i), ui);
```

Output:

```
d: -2147483648 -2147483648
u: 2147483648 2147483648
```

Width and precision

```
int width = 10;
int precision = 4;
double pi = 3.14159265358979323846;
```

```
printf("%*.*f\n", width, precision, pi);
```

needs to become

```
fmt::println("{:{}.{}f}", pi, width, precision);
```

- That's a rotate!

modernize-use-std-print

- https://clang.llvm.org/extra/clang-tidy/checks/modernize/use-std-print.html
- https://github.com/llvm/llvm-project/
  - `clang-tools-extra/clang-tidy/utils/FormatStringConverter.cpp`
  - `clang-tools-extra/clang-tidy/modernize/UseStdPrintCheck.cpp`
  - `clang-tools-extra/test/clang-tidy/checkers/modernize/use-std-print.cpp`

Now to apply it to my logging problem 1

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{i=%03d, s=%20s, u=%#x}\n", i, s, u);
}
```

becomes

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{{i={:03}, s={:>20}, u={:#x}}}\n", i, s, u);
}
```

becomes

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{{{{i={{:03}}, s={{:>20}}, u={{:#x}}}}}}\n", i, s, u);
}
```

Now to apply it to my logging problem 2

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{i=%03d, s=%s, u=%#x}\n", i, s, u);
  LOG("A message that requires no changes\n");
}
```

becomes

```
void f(int i, const std::string &s, uint32_t u) {
  LOG.FMT_SPEC("{{i={:03}, s={}, u={:#x}}}\n", i, s, u);
  LOG.FMT_SPEC("A message that requires no changes\n");
}
```

Now to apply it to my logging problem 3

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{i=%03d, s=%s, u=%#x}\n", i, s, u);
  LOG("A message that requires no changes\n");
}
```

becomes

```
void f(int i, const std::string &s, uint32_t u) {
  LOG("{{i={:03}, s={}, u={:#x}}}\n", i, s, u);
  LOG("A message that requires no changes\n");
}
```

5,000 more lines changed

```
-    LOG("INTERFACE '%s' is WiFi and is an access point\n", m_iface);
+    LOG("INTERFACE '{}' is WiFi and is an access point\n", m_iface);
```

LOG-fixing check not suitable for upstream

- Too specific to BrightSign implementation
- Only about twenty lines of code
- But a hundred lines of lit tests
- github/mikecrowe/clang-tidy-fmt mac/clang-tidy-trace-format branch

Not so fast! A strange crash?

- Unit test failure:
  - uncaught exception
  - `fork(2)`
  - redirecting file descriptors
  - non-blocking file descriptors
  - signals
  - executing NodeJS
  - only reproducible under valgrind
  - the offending call wasn't in the unit test code

Divide and conquer

- Commit each converted file separately

```
git status -s | grep '^ M ' | cut -c4- | while read x; do
    git add "$x"
    git commit -m "FMT_STYLE $x"
done
```

- Then use git bisect to identify the cause.

The cause

Old code

```
template <typename... Args>
inline int catching_printf(FILE *fp, const char *format, Args&&... args)
{
  try {
    return fmt::fprintf(fp, format, std::forward<Args>(args)...);
  }
  catch (const std::exception &e) {
    return fmt::fprintf(fp, "Message '%s' threw '%s'\n", format, e.what());
  }
}
```

New code

```
template <typename... Args>
inline void catching_print(FILE *fp, const char *format, Args&&... args)
{
  try {
    fmt::print(fp, format, std::forward<Args>(args)...);
  }
  catch (const std::exception &e) {
    fmt::print(fp, "Message '{}' threw '{}'\n", format, e.what());
  }
}
```

Format string checking

```
class AlwaysLog {
public:
    template <typename... Args>
    void operator()(std::format_string<Args...> fmt, Args&&...args) {
        std::print(stderr, fmt, std::forward<Args>(args)...);
    }
};
```

Format string error messages

```
<source>: In function 'int main()':
<source>:15:8: error: call to consteval function 'std::basic_format_string<char, const char (&)[6]>("Hello {} {}\012")' is not a constant expression
   15 |     LOG("Hello {} {}\n", "world");
      |     ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from /opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/print:41,
                 from <source>:1:
<source>:15:8:   in 'constexpr' expansion of 'std::basic_format_string<char, const char (&)[6]>("Hello {} {}\012")'
/opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/format:4208:19:   in 'constexpr' expansion of
  '__scanner.std::__format::_Checking_scanner<char, char [6]>::std::__format::_Scanner<char>.std::__format::_Scanner<char>::_M_scan()'
/opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/format:3949:30:   in 'constexpr' expansion of
  '((std::__format::_Scanner<char>*)this)->std::__format::_Scanner<char>::_M_on_replacement_field()'
/opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/format:3982:28:   in 'constexpr' expansion of
  '((std::__format::_Scanner<char>*)this)->std::__format::_Scanner<char>::_M_pc.std::basic_format_parse_context<char>::next_arg_id()'
/opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/format:270:56: error: call to non-'constexpr' function 'void std::__format::__invalid_arg_id_in_format_string()'
  270 |             __format::__invalid_arg_id_in_format_string();
      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
/opt/compiler-explorer/gcc-14.1.0/include/c++/14.1.0/format:218:3: note: 'void std::__format::__invalid_arg_id_in_format_string()' declared here
  218 |   __invalid_arg_id_in_format_string()
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

spdlog

https://github.com/gabime/spdlog

```
class AlwaysLog
{
public:
  template <typename... Args>
  void operator()(spdlog::format_string_t<Args...> fmt, Args&&...args) {
    spdlog::info(std::forward<spdlog::format_string_t<Args...>>(fmt),
                 std::forward<Args>(args)...);
  }
};

AlwaysLog LOG;
int main() {
    LOG("Enter {}:{}\n", __func__, __LINE__);
}
```

1,500 lines left

std::format

```
// Like sprintf but to a new std::string
std::string strprintf(const char *fmt, ...);

std::string GetDescription() const {
  return strprintf("The %d elements of %s", count_, name_);
}
```

becomes

```
std::string GetDescription() const {
  return fmt::format("The {} elements of {}", count_, name_);
}
```

or

```
std::string GetDescription() const {
  return std::format("The {} elements of {}", count_, name_);
}
```

Convert from `absl::StrFormat`

https://abseil.io/docs/cpp/guides/format

```
std::string GetDescription() const {
  return absl::StrFormat("The %d elements of %s", count_, name_);
}
```

becomes

```
std::string GetDescription() const {
  return fmt::format("The {} elements of {}", count_, name_);
}
```

modernize-use-std-format

- https://clang.llvm.org/extra/clang-tidy/checks/modernize/use-std-format.html
- https://github.com/llvm/llvm-project/
  - `clang-tools-extra/clang-tidy/utils/FormatStringConverter.cpp`
  - `clang-tools-extra/clang-tidy/modernize/UseStdFormat.cpp`
  - `clang-tools-extra/test/clang-tidy/checkers/modernize/use-std-format.cpp`

1,000 lines changed

```
Checks: -*,modernize-use-std-format
HeaderFilterRegex: .*
CheckOptions:
  FormatHeader: <fmt/core.h>
  modernize-use-std-format.StrFormatLikeFunctions: strprintf
  modernize-use-std-format.ReplacementFormatFunction: fmt::format
```

```
-    return strprintf("%.3f %s\n", rate, units);
+    return fmt::format("{:.3f} {}\n", rate, units);
```

Replacing Member Functions

```
Checks: -*,modernize-use-std-print
  CheckOptions:
    StrictMode: false
    modernize-use-std-print.PrintfLikeFunctions: Shell::Printf
    modernize-use-std-print.ReplacementPrintFunction: Print
    modernize-use-std-print.ReplacementPrintLnFunction: PrintLn
```

```
  void ShowModel(Shell &shell, const std::string &model_string)
  {
-     shell.Printf("Model: %s\n", model_string.c_str());
+     shell.PrintLn("Model: {}", model_string);
  }
```

7,500 in total!

Macro expansion

- We do want to convert:

```
#include <inttypes.h>
fprintf(stderr,"%" PRIu64 " free clusters\n", f_bfree);
```

- We could perhaps convert:

```
#define LOGIN_USERNAME "brightsign"
strprintf(LOGIN_USERNAME ":%s:%ld:0:99999:7:::\n", /*...*/);
```

- We probably can't automatically convert:

```
#define HIDDEV_PATH_FMT "/dev/usb/hiddev%u"
strprintf(HIDDEV_PATH_FMT, desc->hid_device);
```

Macro expansion, or not

```
printf("Replaceable macros %" PRIu64 " %" PRIu32 "\n", u64, u32);
// CHECK-MESSAGES: [[@LINE-1]]:3: warning: use 'std::println' instead of 'printf'
// CHECK-FIXES: std::println("Replaceable macros {} {}", u64, u32);

#define MYMACRO "d"
printf("Unreplacemable macro %" MYMACRO "\n", s);
// CHECK-MESSAGES: [[@LINE-1]]:3: warning: unable to use 'std::print' instead of 'printf'
//                 because format string contains unreplaceable macro 'MYMACRO'
```

Fixes in templated code

```
#include <catch2/catch_all.hpp>

template<class HASH_TYPE>
void TestMemento(const char *hash_name)
{
    HASH_TYPE hash;
    REQUIRE_THROWS_WITH(hash.FromMemento(""),
                        Equals(strprintf("%s memento too small", hash_name)));
}
```

Making use of AI

- ChatGPT was occasionally quite useful:
  - “How can I tell if the FunctionDecl includes a variable-argument list?”
  - “In this context, how can I tell the difference between a C-style variadic function and a template variadic function?”
  - “Please write a function to go into LLVM that converts a type to its unsigned counterpart.”

Reviewers

- Thanks to the clang-tidy reviewers:
  - Piotr Zegar
  - Carlos Galvez
  - Nathan James
  - Danny Mösch
  - Julian Schmidt
  - Eugene Zelenko
- and the {fmt} and std::format proposal author:
  - Victor Zverovich

Go forth and write clang-tidy checks 1

- Compiling clang-tidy yourself isn't hard
- You don't need to be a language parsing expert
  - Use `clang -Xclang -ast-dump -fsyntax-only test.cpp` to look at the syntax tree
- Write a general-purpose check that can be configured or customised for your use case
- But for a one-off it may not matter
- Sometimes search-and-replace is quicker and easier

Go forth and write clang-tidy checks 2

- The documentation for getting started is good
  - https://clang.llvm.org/extra/clang-tidy/Contributing.html
- Plenty of other checks to look at for inspiration.
- Search the list of AST matchers for likely candidates:
  - https://clang.llvm.org/docs/LibASTMatchersReference.html
- If the AST matcher DSL compiles then it probably does something useful.
- Upstream reviewers are helpful, if sometimes slow and brief.

Resources

Slides can be downloaded from
https://www.mcrowe.com/meetingcpp2024.pdf

Try the new clang-tidy checks at
https://godbolt.org/z/jTPfKz4dx

Questions
mastodon.social/@mikecrowe