Sep 19, 2019

Are We Macro-free Yet?

ZHIHAO YUAN <LICHRAY@GMAIL.COM>

SIMPLEROSE INC

Schedule

Background

The macros that we eliminated: `#if`

The macros that we have not eliminated

The macros that should be prioritized for elimination

Areweyet

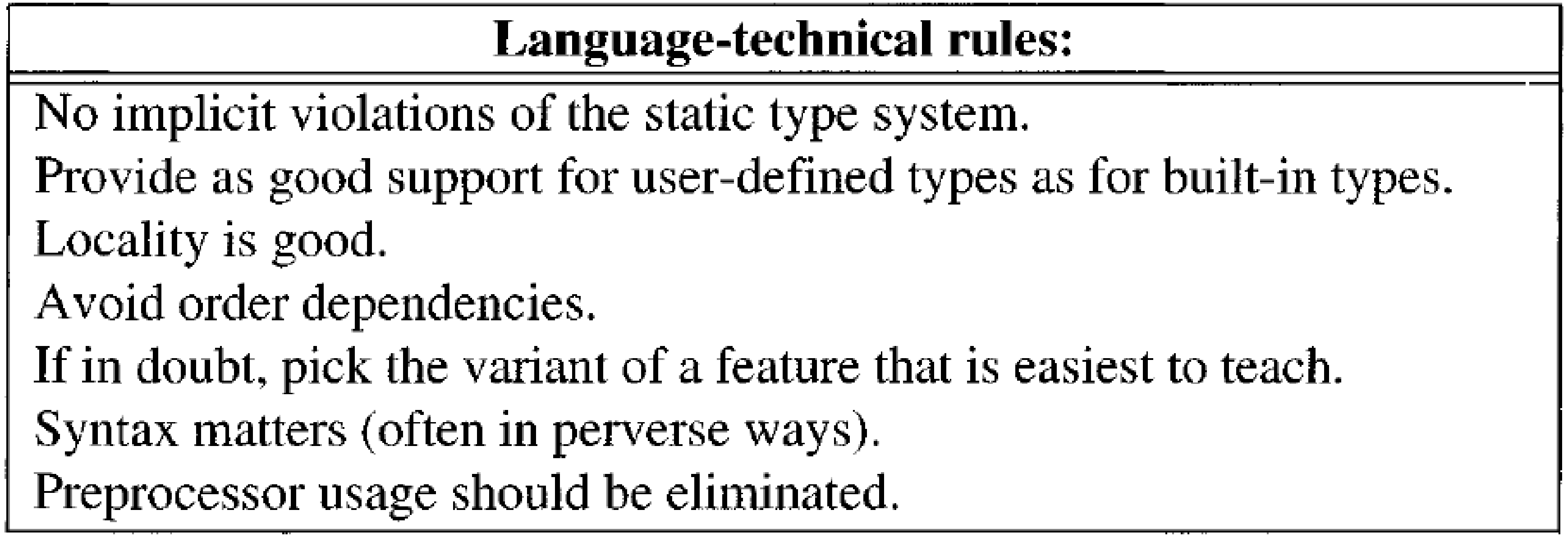
A Mozilla tradition to track top-level progress metrics using “are we” sites.

- Are we web extensions yet? http://arewewebextensionsyet.com/
- Are we Chrome yet? http://arewechromeyet.com/

Rust folks inherited this tradition.

- Are we async yet? https://areweasyncyet.rs/
- Are we web yet? http://www.arewewebyet.org/
- Are we IDE yet? https://areweideyet.com/

Why asking “Are we macro-free yet?”



Language-technical rules:
No implicit violations of the static type system.
Provide as good support for user-defined types as for built-in types.
Locality is good.
Avoid order dependencies.
If in doubt, pick the variant of a feature that is easiest to teach.
Syntax matters (often in perverse ways).
Preprocessor usage should be eliminated.

Ask “Are we macro-free yet,” or ask

```
cublasHandle_t p;
assert(cublasCreate(&p) == CUBLAS_STATUS_SUCCESS);
```

“Why does the handle become uninitialized in Release build?”

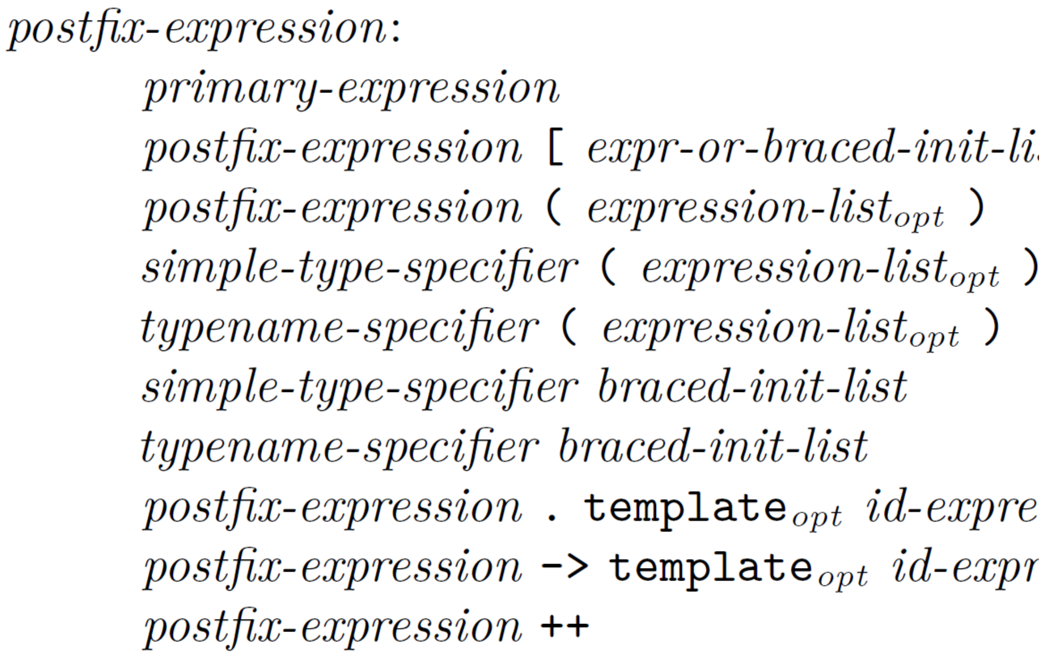
...or ask

```
      #define PRINT(out, a) out << #a " :\n"; out << a;
      PRINT(out, indices);
+     if (matrix.dimensions() != 0)
+         PRINT(out, matrix);
```

“Why did I think people will take a glance at the macro definition?”

Code involving macro isn’t C++

C++ GRAMMAR



postfix-expression:

primary-expression
postfix-expression `[` *expr-or-braced-init-li*
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
typename-specifier `(` *expression-list*$_{opt}$ `)`
simple-type-specifier braced-init-list
typename-specifier braced-init-list
postfix-expression `.` `template`$_{opt}$ *id-expre*
postfix-expression `->` `template`$_{opt}$ *id-expr*
postfix-expression `++`

MY CODE

```
PRINT(out, indices);
if (matrix.dimensions() != 0)
    PRINT(out, matrix);
```

Modern C++ implies no macro

```
# define smart_ptr(Kind, Type, ...)                                  \
    ({                                                               \
        struct s_tmp {                                               \
            CSPTR_SENTINEL_DEC                                       \
            __typeof__(Type) value;                                  \
            f_destructor dtor;                                       \
            …
# define shared_ptr(Type, ...) smart_ptr(SHARED, Type, __VA_ARGS__)
# define unique_ptr(Type, ...) smart_ptr(UNIQUE, Type, __VA_ARGS__)
```

“Modern C” may imply the opposite

A long history of fighting macros

“One of C++’s aims is to make C’s preprocessor redundant because I consider its actions inherently error prone.”

Stroustrup, B. (1994). The Birth of C++. In *The Design and Evolution of C++* (pp. 63-108). Reading, MA: Addison Wesley.

Replace local function-like macros with lambdas

`inline` short functions

Supersede `<tgmath.h>` with function overloading

Alias parameterized types with alias templates

Define constants with (`inline`) `constexpr`

Replace `NULL` with `nullptr`

Repeat code with templates

Replace literal creation macros (INT64_C) with UDL

Standardize attributes such as `[[noreturn]]`

Replace `TYPEOF` with `decltype`

…

What about conditional compilation?

- Why `#if` is bad?

What *constexpr if* statement can do to conditional compilation?

- ◦ Understanding constexpr if statement
- ◦ Scoping conditional compilation

What happens if HAVE_BLAS is a typo?

```
#ifdef HAVE_BLAS
    cblas_daxpy(…);
#else
    std::transform(…);
#endif
```

What this is testing?

```
#if defined(_MSC_VER) && _MSC_VER < 1900
  …some definitions
#endif
```

What this is testing again?

```
#if __cpp_deduction_guides >= 201907L
  …some declarations
#endif
```

Is that still C++ code?

```
#if defined(_WIN32)
    int fd;
    if (_sopen_s(&fd, fn, _O_RDONLY, _SH_DENYWR, 0) == 0)
#else
    if (auto fd = ::open(fn, O_RDONLY))
#endif
        return …;
```

Begging for goto fail

```
#if defined(_WIN32)
    if (bypass_wchar_conversion()) {
        // ...
    } else
#endif
    ok = swritew_b(s, d) and sflush() and
```

Problems with `#if`

There is no guarantee that building all combinations of configurations can reveal a logic error in the conditions

Encouraging testing conditions without semantics

Inviting obscure code structure

My brain is not a preprocessor

What about conditional compilation?

Why `#if` is bad?

What $constexpr\ if$ statement can do to conditional compilation?

- Understanding constexpr if statement
- Scoping conditional compilation

Understanding constexpr if statement

```
template<class T>
bool close_handle(T x)
{
    if constexpr (std::is_same_v<T, int>)  // dependent
        return ::close(x) == 0;
    else
        return ::CloseHandle(x);
}
```

Customize instantiations?

```
template<class T>
bool close_handle(T x);
```

Like partial specializations

```
template<class T, bool = std::is_same_v<T, int>>
bool close_handle(T x);


close_handle<*, true>
close_handle<*, false>
```

[†] Functions don’t have partial specializations.

If the template used to look like this…

```
template<class T, bool = std::is_same_v<T, int>>
bool close_handle(T x)
{
    if constexpr (std::is_same_v<T, int>)
        return ::close(x) == 0;
    else
        return ::CloseHandle(x);
}
```

WARNING: just a way for you to understand. Constexpr-if is not a syntax sugar.

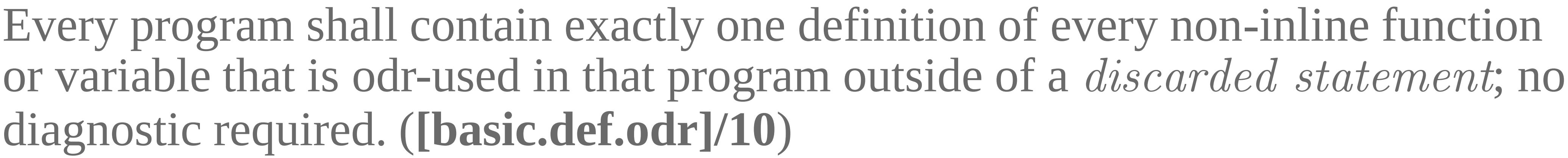
Specializations happening locally

```
template<class T>
bool close_handle<T, true>(T x)
{
    if (true)
        return ::close(x) == 0;
    else
        ~~return ::CloseHandle(x);~~
}
```

With discarded statement

```
template<class T>
bool close_handle<T, false>(T x)
{
    if (false)
        ~~return ::close(x) == 0;~~
    else
        return ::CloseHandle(x);
}
```

Discarded statement (1/2)



Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a *discarded statement*; no diagnostic required. (**[basic.def.odr]/10**)

Implies: A function or a variable that is odr-used inside a discarded statement may have zero definitions.

Such a function or a variable still must be declared, otherwise the name is not introduced, nor the interpretation and semantic properties to come with the name.

Understanding constexpr if statement

```
int close_fd(int fd)
{
    if constexpr (have_iso_conformant_api)  // non-dependent
        return _close(fd);
    else
        return ::close(fd);
}
```

If this used to be a template…

```
template<bool = have_iso_conformant_api>
int close_fd(int fd)
{
    if constexpr (have_iso_conformant_api)
        return _close(fd);
    else
        return ::close(fd);
}
```

Customized with explicit specializations

```
template<bool = have_iso_conformant_api>
int close_fd(int fd);
```

We can explicitly define the following specializations:

```
close_fd<true>
close_fd<false>
```

Locally

```
template<>
int close_fd<true>(int fd)
{
    if (true)
        return _close(fd);
    else
        ~~return ::close(fd);~~
}
```

With discarded statement

```
template<>
int close_fd<false>(int fd)
{
    if (false)
        ~~return _close(fd);~~
    else
        return ::close(fd);
}
```

What about conditional compilation?

Why `#if` is bad?

What $constexpr\ if$ statement can do to conditional compilation?

- Understanding constexpr if statement
- Scoping conditional compilation

Scoping conditional compilation

1. Replacement within function definitions
2. Replacing class definitions

Replacement within function definitions

```
void daxpy(double a, span<double const> x, span<double> y)
{
      #ifdef HAVE_CBLAS
          cblas_daxpy(…);
      #else
          std::transform(…);
      #endif
}
```

Interface

Implementation

Test variables, not macros

```
void daxpy(double a, span<double const> x, span<double> y)
{
      if constexpr (have_cblas)
          cblas_daxpy(…);
      else
          std::transform(…);
}
```

Hard error if have_cblas is never defined

Breaking it down: condition

build_config.h:

```
    constexpr bool have_cblas = ??;
```

Introduce variables without macros

build_config.h.in:

```
    constexpr bool have_cblas = @HAVE_CBLAS@;
```

Let build systems solve build problems.

CMake example

```
find_package(BLAS)
if(BLAS_FOUND)
    set(HAVE_CBLAS true)
else()
    set(HAVE_CBLAS false)
endif()

configure_file(build_config.h.in
               build_config.h @ONLY)
```

build_config.h.in:

```
constexpr bool have_cblas = @HAVE_CBLAS@;
```

build_config.h if BLAS not found:

```
constexpr bool have_cblas = false;
```

Unconditionally introduce the names

```
#include <algorithm>   // for std::transform

extern "C" void cblas_daxpy(int n, double alpha,
                            double const* x, int incx,
                            double* y, int incy);
```

Conditional operations

```
#ifdef HAVE_CBLAS
    cblas_daxpy(…);
#else
    std::transform(…);
#endif
```

Conditional declarations?

```
#ifdef HAVE_ZLIB
    gzFile fp = gzopen(filename, "r");
#else
    FILE* fp = fopen(filename, "r");
#endif
```

Immediately invoked lambdas

```
auto fp = [&] {
    if constexpr (have_zlib)
        return gzopen(filename, "r");
    else
        return fopen(filename, "r");
}();
```

[‡] Declaration of gzopen is available on zlib website.

Discarded statement (2/2)

If the declared return type of the function contains a placeholder type, the return type of the function is deduced from non-discarded `return` statements, if any, in the body of the function. (**[dcl.spec.auto]/3**)

Implies: Discarded statements do not contribute to return type deduction.

Limitation of constexpr-if in practice

```
int64_t get_file_size(char const* filename)
{
#if defined(_WIN32)
    struct _stat64 st;
    _stat64(filename, &st);
#else
    struct stat st;
    ::stat(filename, &st);
#endif
```

Definition of `struct _stat64` is required to define variables

When a complete type is required but conditionally available

~~Define the type by yourself~~
- ODR violation when including the corresponding header

Rethink about the function – can it be deemed **disjointed** implementations?
- If so, we can replace the implementations with build systems

Breaking it down: Translation units

src/win32.cc:

```
int64_t
get_file_size(char const* filename)
{
    struct _stat64 st;
    _stat64(filename, &st);
    return st.st_size;
}
```

src/posix.cc:

```
int64_t
get_file_size(char const* filename)
{
    struct stat st;
    ::stat(filename, &st);
    return st.st_size;
}
```

CMake example

```
if(WIN32)
    list(APPEND mylib_srcs src/win32.cc)
else()
    list(APPEND mylib_srcs src/posix.cc)
endif()

target_sources(mylib ${mylib_srcs})
```

Scoping conditional compilation

1. Replacement within function definitions
2. Replacing class definitions

Replacing class definitions

```
struct DirStreamCore {
#if defined(_SYS_MSVC_) || defined(_SYS_MINGW_)
  Mutex alock;                           ///< attribute lock
  ::HANDLE dh;                           ///< directory handle
  std::string cur;                       ///< current file
#else
  Mutex alock;                           ///< attribute lock
  ::DIR* dh;                             ///< directory handle
#endif
};
```

typical reason: data members are different

“High-level components should not depend on low-level components”

```
struct DirStreamCore {
#if defined(_SYS_MSVC_) || defined(_SYS_MINGW_)
  Mutex alock;
  ::HANDLE dh;
  std::string cur;
#else
  Mutex alock;
  ::DIR* dh;
#endif
};
```

Interface?
Implementation?

Answer: Dependency inversion

“High-level components should not depend on low-level components.”

- Remove low-level dependency from class definition – **PImpl**

“Both should depend on abstractions.”

- Create an implicit, non-virtual interface (abstraction) that allows substitution of implementations – **Type erasure**

Before

```
class DirStream {
 public:
  explicit DirStream();
  ~DirStream();
  bool open(const std::string& path);
  bool close();
  bool read(std::string* path);
 private:
  void* opq_;
};
```

```
bool DirStream::close() {
#if defined(_SYS_MSVC_) || defined(_SYS_MINGW_)
  DirStreamCore* core = (DirStreamCore*)opq_;
  …
```

C-style “Type erasure”

After

```
class DirStream {
 public:
  bool open(const std::string& path)  { return this_->open(path); }
  bool close()                        { return this_->close(); }
  bool read(std::string* path)        { return this_->read(path); }
 private:
  struct DirStreamInterface {…};
  template<class T>
  struct DirStreamCore final : DirStreamInterface {…};
  std::unique_ptr<DirStreamInterface> this_;
};
```

PImpl

include/mylib/win32dirstreamcore.h:

```
struct Win32DirStreamCore {
  bool open(const std::string& path);
  bool close();
  bool read(std::string* path);
 private:
  class impl;
  unique_ptr<impl> impl_;
};
```

include/mylib/posixdirstreamcore.h:

```
struct PosixDirStreamCore {
  bool open(const std::string& path);
  bool close();
  bool read(std::string* path);
 private:
  class impl;
  unique_ptr<impl> impl_;
};
```

Type erasure

Tomorrow afternoon,

Back to Basics: Type Erasure

from Arthur O’Dwyer, 13:30 - 14:30.

Flexibility of dependency inversion

1. Build target (OS, Toolchain, etc.) bonded implementations
2. Selecting a single implementation at build time
3. Selecting implementation at runtime from a set of implementations determined at build time
4. Test a set of implementations determined at build time

Build target bonded implementations

The choice of implementation is implied for a given target.

```
using DirStreamCoreImpl =
    std::conditional_t<have_win32_api,
                       Win32DirStreamCore,
                       PosixDirStreamCore>;
```

Select one implementation at build time

```
enum class backend { tbb, openmp, cuda };
…
template<backend v>
using select_backend = typename select_backend_imp<v>::type;
// build_config.h.in
constexpr backend backend_to_use = @MYLIB_BACKEND@;
# CMakeLists.txt
set_property(CACHE MYLIB_BACKEND PROPERTY STRINGS tbb openmp cuda)
```

CUDA & Conditionally available toolchains

Heterogenous toolchains are flexible

Apply on optional libraries rather than optional translation units

```
add_library(mylib …)
if(CMAKE_CUDA_COMPILER)   # check_language(CUDA)
    add_library(mylib-parallel …)
    target_link_libraries(mylib mylib-parallel)
endif()
```

Share your PImpl header in both libraries

Determine a set of implementations

So that we can select from them at runtime.

```
// a type list
using implementations = std::conditional_t<
    have_cuda_toolkit,
    std::tuple<tbb_impl, openmp_impl, cuda_impl>,
    std::tuple<tbb_impl, openmp_impl>>;
```

Run unit tests on implementations determined at build time

doctest[3] example:

```
TEST_CASE_TEMPLATE_DEFINE("simple", T, test_simple)
{
    auto x = mylib::algorithm_backend(in_place_type<T>);
    REQUIRE(…);
}
DOCTEST_TEMPLATE_APPLY(test_simple, mylib::implementations);
```

More macros to kill?

Include guards

Logging

Metadata macros (e.g. `Q_OBJECT`)

Unit testing framework

Include guards

Least harmful macros. Visually do not interact with code.

Modules will eliminate them one day.

A typical logging macro

```
// LOG_F(2, "Only logged if verbosity is 2 or higher: %d", some_number);
#define VLOG_F(verbosity, ...)                             \
        ((verbosity) > loguru::current_verbosity_cutoff()) \
            ? (void)0                                      \
            : loguru::log(verbosity, __FILE__, __LINE__, __VA_ARGS__)
```

When to evaluate?

Macro-free logging

Some users want to optionally track file names and line numbers

- C++20 `std::source_location` will address that

Some users may want lazy evaluation of formatting arguments

- This is not the mental model when we are reading

```
warning("Only logged if verbosity is high: %d", fp.fileno());
```

- A `std::format` (C++20) based macro-free logging framework would behave similar to `spdlog` and Python standard library's `logging` module

Metadata macros

Complicates codegen but less so on code reading

What static reflection meant to replace:

- iterating over struct fields, enum members

What metaclasses (generative programming) meant to replace:

- generating declarations

Disclaimer: I’m not promising anything. Watch Andrew Sutton’s talks.

Macro-free unit testing framework

This afternoon,

Next generation unit testing using static reflection[5]

from Manu Sánchez, 14:00 - 15:00.

What macros to eliminate first?

The macros that interleave with program logic in any form

- conditional code blocks, token soup
- function-like or object-like macros that substitute into expressions

The macros that hijack interface with implementation details

- don’t take over build systems’ job
- consider a better design

Think about how to migrate away before introducing a macro

Questions?

@lichray

Demo

lichray/macrofree-demo

CAST

1. Smart pointers for the (GNU) C programming language https://github.com/Snaipe/libcsptr
2. Kyoto Cabinet: a straightforward implementation of DBM https://github.com/cloudflarearchive/kyotocabinet
3. doctest: The fastest feature-rich C++11/14/17/20 single-header testing framework for unit tests and TDD https://github.com/onqtam/doctest
4. loguru: A lightweight C++ logging library https://github.com/emilk/loguru
5. unittest: C++ unit testing and mocking made easy https://github.com/Manu343726/unittest