
zühlke
empowering ideas
“Just Switch the Compiler,”
they said.
Arne Mertz

Arne Mertz
Software Engineer at Zühlke
(mostly embedded projects)
Working with C++ for ca. two decades
Trainer for C++ and maintainable code

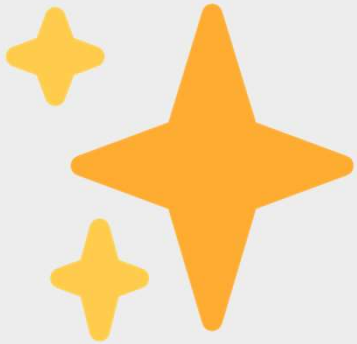
Agenda

- Reasons to switch the compiler
- Steps to switch, pitfalls to expect
- Strategies to avoid those pitfalls

Why switch?

Access to new C++ standards

zühlke
empowering ideas



New target

- Windows, Linux, Mac
- Intel, ARM
- Embedded targets

Access to other features

- Compiler or toolchain features, e.g.
  - Optimizer, static analysis, instrumentation for runtime analysis
  - Features of the build system, e.g. code generation, support for larger projects
  - IDE features, e.g. refactoring support, AI companion
- Changing dev machines and other infrastructure
  - E.g. not every compiler works on every OS

Get rid of bugs and quirks

- In the compiler or in the toolchain it depends on, e.g.
  - Dependency on an outdated operating system
  - Bugs in the compiler, debugger, IDE


zühlke
empowering ideas
“Well, then Just Switch the Compiler”


ONE DOES NOT SIMPLY
SWITCH TO ANOTHER COMPILER

No simple “switch”

- Multiple steps required
  - It can be a lengthy process
- Usually requires using both compilers in parallel
  - Verify changes necessary for the new compiler in the old system
  - Continued feature development

→ Switching or adding is mostly the same

Infrastructure Setup

zühlke
empowering ideas

New development environment

Just installing the compiler may not be enough

- Compilers depend on toolchains ad vice versa
  - ⛔ Potential roadblock, e.g. if the toolchain is dictated by vendors

- What about licenses? (Software, hardware)
  - Financial, legal, and political implications
  - Involvement of IT department

Toolchain

zühlke
empowering ideas

- Find, install, configure the necessary and desired tools:
  - Build system
  - Code formatting
  - Static & dynamic analysis
  - Code generators
  - Package manager
  - Remote debugger
  - … and more

New development environment

- New installation may collide with the old one
  - E.g. installing different versions of the same tool
- Consider the possibilities for separate installations
  - Native installation
  - VM
  - Containerized build
  - WSL

Milestone!

Barebone compiler & toolchain installation

zühlke
empowering ideas

```
1   #include <iostream>
2
3   int main() {
4     std::cout << "Hello, world!\n";
5   }
```

“Works on my machine” edition

New CI environment

- Like development environment, but +1 in difficulty
  - No access to hardware
  - Virtual machines
  - Often controlled by IT department (⛔)

Document build environments

Local and CI

- Repeatable setup
  - Detailed documentation or setup script
  - For old and new environment
  - Consider to have it version controlled with the code base
- Later joiners will thank you
- You'll be glad when you restore a 2025 build a few years from now

Milestone!

Development environment

zühlke
empowering ideas

```cpp
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";
}
```

“Works on your machine, too” edition

Dependencies

Dependencies

Libraries and frameworks

- Check that 3rd party libraries are compatible
  - Rebuild them locally
  - Set up CI build
  - Buy closed source binaries
- Make them available to team and CI
  - E.g. upload to package repository

Dependencies

Replacing libraries

- Your library/framework may not be available for the new compiler or target
  - E.g. windows.h vs. Linux system headers
  - Specialized libraries
- Find alternatives
  - Preferably with similar design, architecture, interfaces
  - Alternative may be the new standard library
  - ⛔ Potential road block: Does an alternative even exist?
- Decide:
  - Full replacement (old and new compiler)
  - Using different libraries depending on compiler

Conditional compilation

zühlke
empowering ideas

- Check what to include

```
1  #ifdef _WIN32
2  #  include <windows.h>
3  #else #ifdef _WIN32
4  #  include <fcntl.h>
5  #  include <unistd.h>
6  #endif #ifdef _WIN32 #else
```

- … and which functions to call

Conditional linkage

CMake example

zühlke
empowering ideas

- Check what to link into your libraries
- … and which libraries to link into your executables

```
12  if (MSVC)
13      target_sources(my_lib some_impl_win.cpp)
14      target_link_libraries(my_exe thisLib)
15  else()
16      target_sources(my_lib some_impl_linux.cpp)
17      target_link_libraries(my_exe thatOtherLib)
18  endif()
```

Conditional compilation/linkage

- Consider using project specific #defines, Cmake variables, etc.
  - → easier to find your customization points
- Consider not having compileable else branches
  - → clear error instead of hard to debug errors or behavior

```
6   #ifdef MYPROJECT_COMPILER_MSVC
7   using SysImpl = MSVCImpl;
8   #elif MYPROJECT_COMPILER_GCC_ARM || MYPROJECT_COMPILER_CLANG
9   using SysImpl = GNUImpl;
10  #else #elif MYPROJECT_COMPILER_GCC_ARM || MYPROJECT_COMPILER_CLANG
11  #error "unknown/undefined project compiler"
12  #endif #elif MYPROJECT_COMPILER_GCC_ARM || MYPROJECT_COMPILER_CLANG #else
```

Dependencies

Replacing libraries

- Encapsulate dependencies
  - Fewer points for conditional compilation/linking
  - Unit tests can be written dependency agnostic

- Contain dependencies
  - Separation of concerns, e.g. separate GUI code from business logic
  - Fewer tests to be run with dependencies

Dependencies

Containing dependencies

- Containing dependencies after the fact can be a lengthy process
  - Make a conscious design/architecture decision where to use them

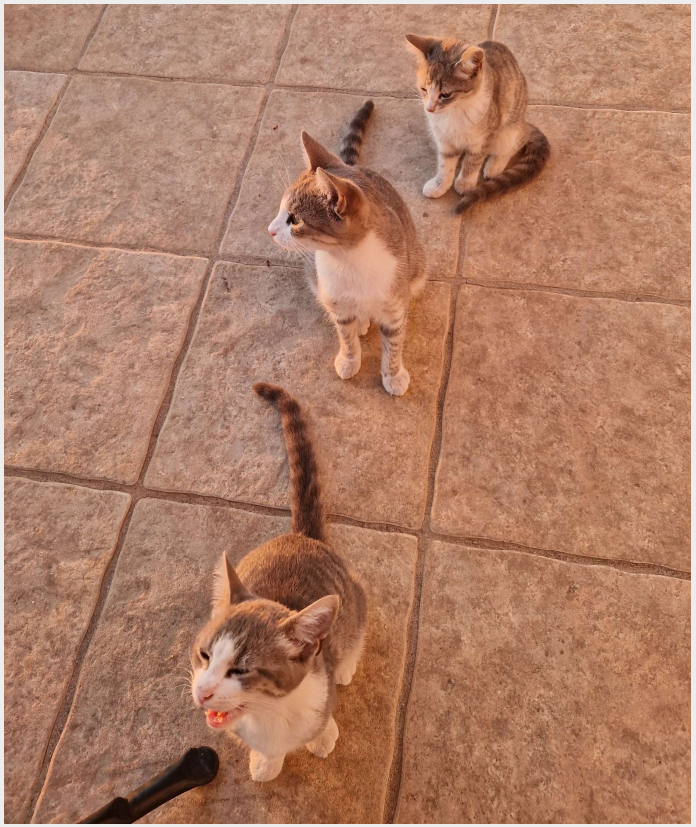
- Example: GUI framework
  - Do we allow some logic in the GUI layer or is it strictly separated?
  - When using Model-View-Controller, do we restrict use of the GUI framework
    - To the view only
    - To View and Controller
    - Or more?

Milestone!

Dependencies

zühlke
empowering ideas

Everything is in place to port the project



Compile the project

zühlke
empowering ideas

Call the compiler

From your build system

- Integrate the new compiler into your build scripts
  - Command line options
  - #defines for conditional compilation
  - … and more

- Does it compile immediately?
  - Probably not…

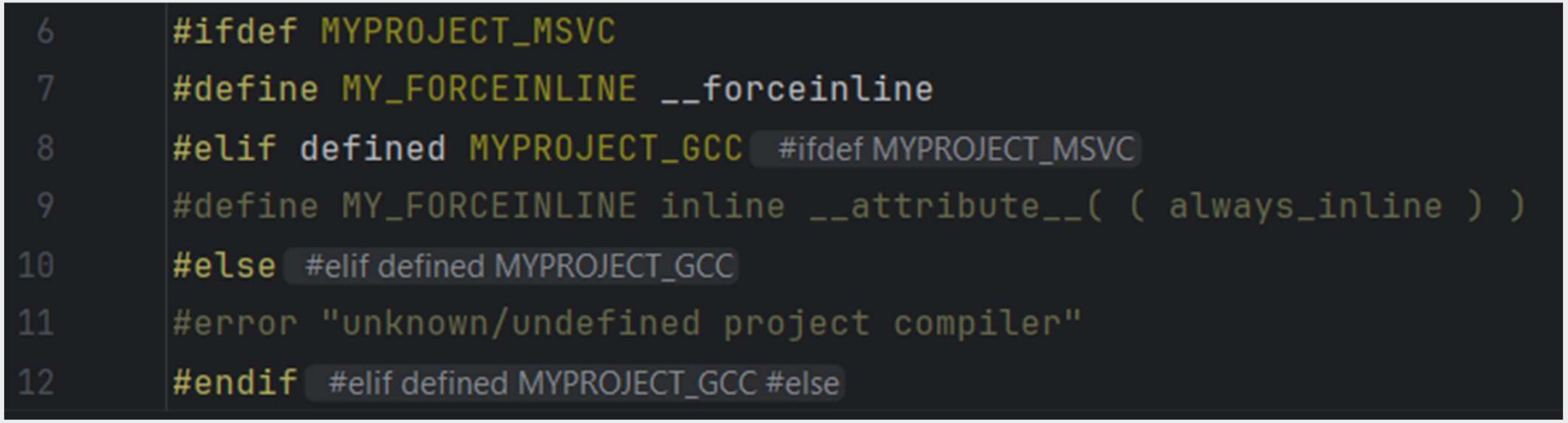
Compiler specific extensions

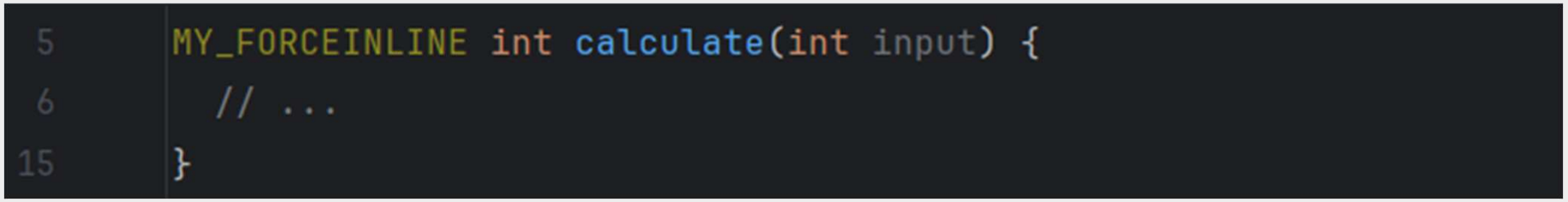
Example

- `__attribute__((always_inline))` vs. `__forceinline`
  - Many similar cases with nonstandard extensions
  - Often available on multiple compilers
  - May have slightly different semantics

Compiler specific extensions

Attribute macro example





Compiler specific extensions

DLL import/export example

- DLLs and shared objects are platform specific extensions

```
4   #ifdef MYPROJECT_OS_WINDOWS
5   # ifdef COMPILING_SOMELIB_DLL
6   #  define SOMELIB_EXPORT __declspec(dllexport)
7   # else #ifdef COMPILING_SOMELIB_DLL
8   #  define SOMELIB_EXPORT __declspec(dllimport)
9   # endif #ifdef COMPILING_SOMELIB_DLL #else
10  #else #ifdef MYPROJECT_OS_WINDOWS
11  # define SOMELIB_EXPORT
12  #endif #ifdef MYPROJECT_OS_WINDOWS #else
```

```
19  class SOMELIB_EXPORT SomeClass
```

Compiler specific extensions

Code breakpoints

- Check if your dependencies/extensions are available in the standard
  - std::breakpoint()

Compiling with new standards

Beware of obsolete/changed features

https://en.cppreference.com/w/cpp/17.html

Removed features

- `std::auto_ptr`,
- deprecated function objects,
- `std::random_shuffle`,
- `std::unexpected`,
- the obsolete `iostreams` aliases,
- trigraphs,
- the `register` keyword,
- `bool` increment,
- dynamic exception specification

Deprecated features

- `std::iterator`,
- `std::raw_storage_iterator`,
- `std::get_temporary_buffer`,
- `std::is_literal_type`,
- `std::result_of`,
- all of `<codecvt>`

Implementation-defined behavior

At compile time

- Index of Implementation-defined behavior: 4+ pages in N4950 (C++23)
  - Includes standard library
- #include may find different files
- Things your code uses might not exist on the new compiler
- Things your code uses might have different types or sizes
  - Effects on narrowing conversions, `sizeof( )`, etc.

Unspecified behavior

At compile time

- Whether names from <c????> headers are first declared within the global namespace scope and are then injected into namespace std by explicit using-declarations
  - e.g. `abs` vs `std::abs`
  - `size_t` vs `std::size_t`

Warnings

- Different compilers have different sets of warnings
  - The new compiler may emit warnings that the old one did not
  - With -Werror that means compile errors
- Ignore at your own peril
  - Turning off -Werror risks delaying compile errors to runtime errors
  - Turn off specific warnings after careful investigation (for now)
- Early prevention: static analyzers
  - More exhaustive sets of warnings


Milestone!
IT COMPILES!
zühlke
empowering ideas
IT COMPIIILES!!
imgflip.com

Run the project

Undefined Behavior

- Posterchild of “anything can happen”

- Code with UB often “just works” as intended
  - At least on the previous compiler
  - You may not know you rely on it
  - Be prepared for surprises

Implementation-defined Behavior

The runtime part

- Values may differ
  - type_info::name()
  - source_location::current
  - <exception>::what()
  - std::hash
- Don’t base logic on implementation defined values
- Log-based tests may need to be updated

Implementation-defined Types

CAN message

zühlke
empowering ideas

```
4   #pragma pack(1)
5
6   struct MotorCommand {
7     unsigned char flags;
8     uint16_t speed;
9     long positionX;
10    long positionY;
11  };
```

- The size of integral types like long is implementation-defined

→ The message has wrong length, receiver does not get correct data

Implementation-defined Types

Runtime bug

zühlke
empowering ideas

```
5   constexpr auto suffixDelimiter = "--";
6   constexpr auto delimLength = 2u;
7
8   std::string getSuffix(std::string const& name) {
9     unsigned const pos = name.find( s: suffixDelimiter);
10    if (pos == std::string::npos) { return ""s; }
11    return name.substr(pos+delimLength);
12  }
```

- Static analysis (clang) warns about "always false" comparison
- Whether unsigned is large enough to hold it is implementation-defined
- C++-Builder 32 bit: OK;
- C++-Builder 64 bit: `sizeof(npos) > sizeof(unsigned)`

  `name.size() < pos < npos` → `std::out_of_range`

Implementation-defined bug (CAN msg)

Implementation-defined Types

Compiler-dependent UB

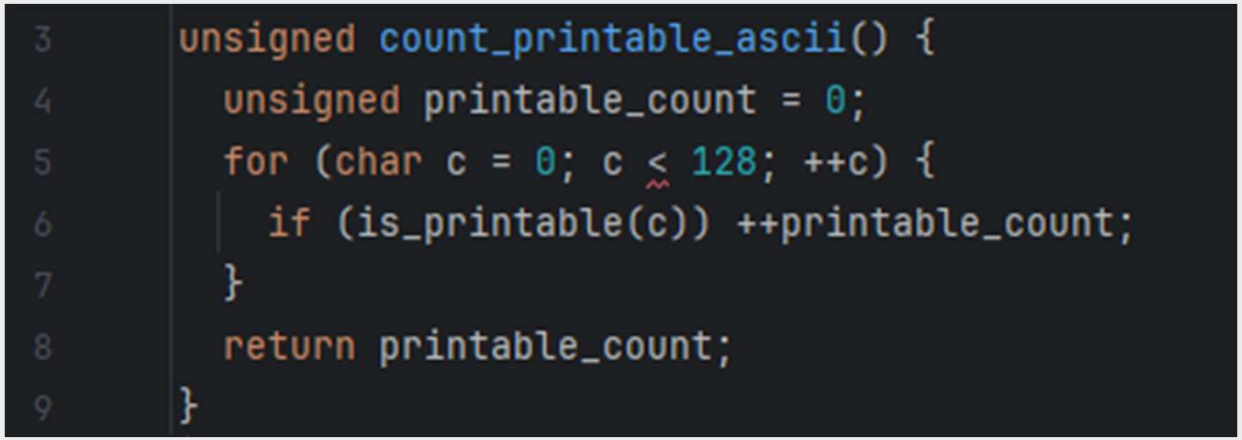
zühlke
empowering ideas

```
4   void uart_putc(char);
5
6   void print_all_ascii() {
7       for (char c = ' '; c < 128; ++c) {
8           uart_putc(c);
9       }
10  }
```

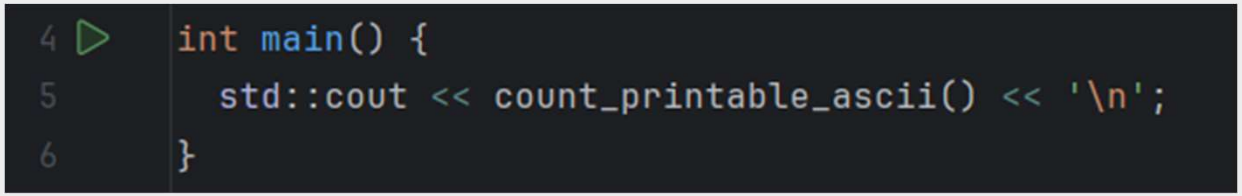
- **Whether char is signed** is implementation-defined
- When it is, it overflows after 127 (UB) (clang warns with -Wall, GCC with -Wextra)
- … and leads to an endless loop (more UB)

Implementation-defined Types

Compiler-dependent UB



```
unsigned count_printable_ascii() {
  unsigned printable_count = 0;
  for (char c = 0; c <= 128; ++c) {
    if (is_printable(c)) ++printable_count;
  }
  return printable_count;
}
```



```
int main() {
  std::cout << count_printable_ascii() << '\n';
}
```

- Clang 21.1.0: calls whatever is implemented right after `count_printable_ascii` - if the optimizer sees the definition of `is_printable`
- E.g. prints 2523119872 – or segfaults

Prevent IB/UB surprises

- Fix warnings *before and after* you switch
  - Crank up your compiler warnings (-Wall -Wextra -Wpedantic -Werror)
  - Use static analysis

- Use UB sanitizer etc.
  - With old and new compiler, if possible

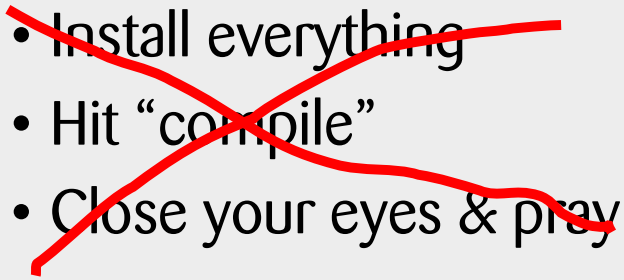
Strategies

Research first

- Verify that the new compiler and toolchain are available for your systems
  - Local and CI
  - Consider VMs, Docker, …
- Verify compatibility or existence of alternatives for libraries
- Find out which language extensions your code uses
  - Find alternatives or workarounds for the new compiler
- Contact your IT and legal departments, if necessary

Make a plan

"Just Switch the Compiler"



- Install everything
- Hit "compile"
- Close your eyes & pray

zühlke
empowering ideas

Make a plan

Define and estimate a roadmap with all needed steps

- Prepare your code base for the switch
  - Encapsulate libraries and extensions
  - Fix all the warnings
  - Refactor build scripts as needed
- Verify and document new infrastructure setup
- Compile and run unit tests
  - Fewer dependencies, less debugging
- Work your way up to full integration


Questions? Comments!
zühlke
empowering ideas

zühlke
empowering ideas

Thank you  Let‘s talk!

 Simplify C++! – www.arne-mertz.de

 @arne_mertz@mastodon.social

 arne.mertz@zuehlke.com

 #include<C++> Discord (includecpp.org)

Emojis by emojione.com