# Clang Tooling

Qt World Summit 2019

Presented by Kevin Funk

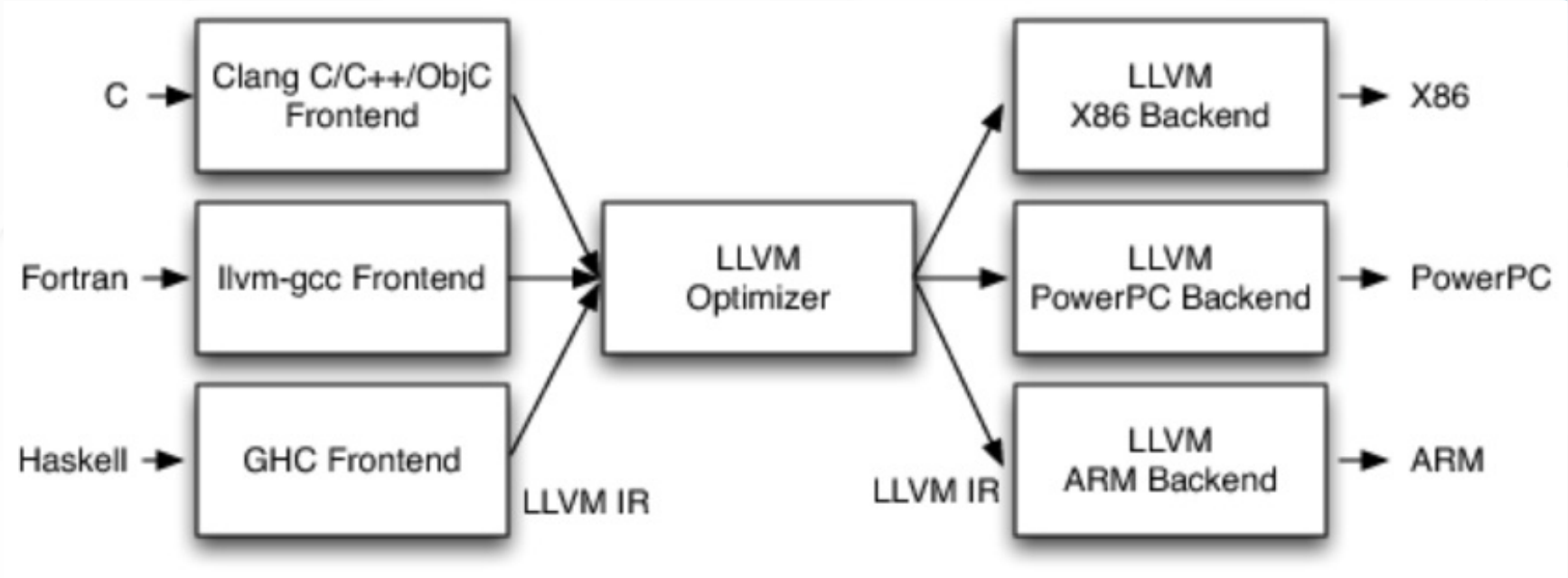
- **Clang Tooling**
  - LLVM/Clang Overview
  - Clang compiler frontend (static analysis)
  - Clang Static Analyzer (static analysis)
  - clang-tidy (static analysis / linter / refactoring)
  - Clazy (static analysis, Qt centric)
  - clang-format (coding style)
  - Clang Sanitizers (dynamic analysis)

- **LLVM/Clang Overview**
- Clang compiler frontend (static analysis)
- Clang Static Analyzer (static analysis)
- clang-tidy (static analysis / linter / refactoring)
- Clazy (static analysis, Qt centric)
- clang-format (coding style)
- Clang Sanitizers (dynamic analysis)

# LLVM vs. Clang

- LLVM
  - Formerly *Low Level Virtual Machine*
  - Set of compiler and toolchain technologies in C++
  - Designed for compile-time, link-time, run-time optimization of programs
    - for arbitrary programming languages
- Clang
  - Compiler front end for C, C++, Objective-C and Objective-C++
    - and nowadays CUDA and OpenCL
  - Uses LLVM as its backend
  - Part of its releases since LLVM 2.6.

# LLVM's implementation of three-phase design



# Clang Highlights

- Expressive diagnostics and *fix-it* hints for warnings and errors
- Library-based architecture, API allows for extensive tooling
- Allows integration with IDEs (→ *QtCreator*, *KDevelop*, Language Server Protocol*, ...)*
- Non-restrictive *BSD license*
- *highly* active and skilled community around it
  - large involvement of big players (Google/Samsung/Apple/TI/...)
- ...
- Interesting overviews:
  - Features: http://clang.llvm.org/features.html
  - Comparison to other compilers: https://clang.llvm.org/comparison.html

# Why is LLVM/Clang so popular?

- It's a real-world, production quality compiler
- Hackable code base, extensive documentation and help from community
- Accessible public APIs for ...
  - Initiating code parsing
  - Traversing the *Abstract Syntax Tree* (AST) of a compilation unit
  - Doing code transformations (refactorings)
  - (and much more)
    - hint: **tooling!**

# Structure of the talk

- *Clang* is not just the compiler binary, there a lot more individual tools:
  - Clang Static Analyzer (scan-build)
  - clang-tidy
  - clang-format
  - ...
- All these tools depend on the Clang libraries
- Next sections will discuss each tool

- LLVM/Clang Overview
- **Clang compiler frontend (static analysis)**
- Clang Static Analyzer (static analysis)
- clang-tidy (static analysis / linter / refactoring)
- Clazy (static analysis, Qt centric)
- clang-format (coding style)
- Clang Sanitizers (dynamic analysis)

# Term: Static Analysis

- Static code (program) analysis:
  - is the analysis of computer software that is performed without executing programs
- on the contrary: analysis performed on executing programs is known as *dynamic analysis*

# What do we want to catch?

- Catch code segments that are valid C++, but (probably) not what you meant to write.
  - typos
  - unintentional operator usage
  - unintentional implicit conversion
  - Potential data over/underflows
  - tautological comparisons
  - dead code
  - *a lot more ...*
- **... at compile time!**

# Clang compiler: Examples

# Clang compiler: Examples

**Example Code (clang_warnings_01.cpp)**

```
1 #include <cstddef>
2
3 int main(int, char**)
4 {
5     std::size_t bigstuff = -1;
6     return bigstuff < 0;
7 }
```

# Clang compiler: Examples

**Example Code (clang_warnings_01.cpp)**

```cpp
#include <cstddef>

int main(int, char**)
{
    std::size_t bigstuff = -1;
    return bigstuff < 0;
}
```

```
$ clang++ -Weverything clang_warnings_01.cpp
clang_warnings_01.cpp:6:28: warning: implicit conversion changes signedness:
'int' to 'std::size_t' (aka 'unsigned long') [-Wsign-conversion]
    std::size_t bigstuff = -1;
                ~~~~~~~~   ^~
clang_warnings_01.cpp:7:21: warning: result of comparison of unsigned expression
false [-Wtautological-unsigned-zero-compare]
    return bigstuff < 0;
           ~~~~~~~~ ^ ~
2 warnings generated.
```

# Clang compiler: Examples (cont'd)

# Clang compiler: Examples (cont'd)

**Example Code (clang_warnings_02.cpp)**

```
1 int main(int argn, char**)
2 {
3 if (argn = 1)
4     return -1;
5 }
```

# Clang compiler: Examples (cont'd)

**Example Code (clang_warnings_02.cpp)**

```
1 int main(int argn, char**)
2 {
3 if (argn = 1)
4     return -1;
5 }
```

```
$ clang++ -Weverything clang_warnings_02.cpp
clang_warnings_02.cpp:5:10: warning: using the result of an assignment as a
condition without parentheses [-Wparentheses]
if (argn = 1)
    ~~~~~^~~
clang_warnings_02.cpp:5:10: note: place parentheses around the assignment to
silence this warning
if (argn = 1)
         ^
    (       )
clang_warnings_02.cpp:5:10: note: use '==' to turn this assignment into an
equality comparison
if (argn = 1)
         ^
         ==
1 warning generated.
```

# Example 3

# Example 3

**Example Code (clang_warnings_03.cpp)**

```
1 /**
2 * @param foo some number
3 * @return some number
4 */
5 void myfunc(int num)
6 {
7 }
8
9 int main() {}
```

# Example 3

**Example Code (clang_warnings_03.cpp)**

```
1 /**
2 * @param foo some number
3 * @return some number
4 */
5 void myfunc(int num)
6 {
7 }
8
9 int main() {}
```

```
$ clang++ -Weverything clang_warnings_03.cpp
clang_warnings_03.cpp:3:5: warning: '@return' command used in a comment that is
attached to a function returning void [-Wdocumentation]
 * @return some number
   ~^~~~~~~~~~~~~~~~~
clang_warnings_03.cpp:2:11: warning: parameter 'foo' not found in the function
declaration [-Wdocumentation]
 * @param foo some number
          ^~~
clang_warnings_03.cpp:2:11: note: did you mean 'num'?
 * @param foo some number
          ^~~
          num
```

# Verdict: Static analysis via Clang compiler

- These warnings are already **built-in** to the clang compiler
  - No need to run extra tools
  - No need to re-parse source for analysis
- Useful for detecting simple control flow / data flow issues
- *-Weverything* is too noisy (i.e. enables *-Wc++98-compat*)
  - instead need to explicitly enable wanted checks
  - *useful* checks:

```
-Wsign-conversion -Wall -Wextra -Wc++0x-compat
-Winline -pedantic -Wredundant-decls -Wloop-analysis -Wstring-compare -Wshadow
-Wunused-variable -Wundef -Wnon-virtual-dtor -Wdocumentation -Wshorten-64-to-32
-Wused-but-marked-unused -Wdisabled-macro-expansion -Wparentheses
-Wsometimes-uninitialized -Wconditional-uninitialized -Wfloat-equal -Wswitch-enum
-Warray-bounds -Wcovered-switch-default -Wunreachable-code
-Wnon-literal-null-conversion -Wtautological-compare -Wcovered-switch-default
```

- LLVM/Clang Overview
- Clang compiler frontend (static analysis)
- **Clang Static Analyzer (static analysis)**
- clang-tidy (static analysis / linter / refactoring)
- Clazy (static analysis, Qt centric)
- clang-format (coding style)
- Clang Sanitizers (dynamic analysis)

# Clang Static Analyzer

- Detects bugs without executing the program
- Deeper data/control flow analysis than Clang compiler
  - similar to cppcheck and Coverity
- Slower than compilation
- Higher rate of false positives
- Runs from the command-line (*scan-build*), results in a webview

# Clang Static Analyzer (cont'd)

- Checkers:
  - Core Checkers
  - C++ Checkers
  - Dead Code Checkers
  - Security Checkers
  - Unix Checkers
- List of available checkers: `scan-build -h`

Run on single file (with *text* output):

**Example Code (clang_sa_warnings_01.cpp)**

```
1 int main()
2 {
3     int *x = new int;
4 }
```

```
$ clang++ --analyze -Xanalyzer -analyzer-output=text clang_sa_warnings_01.cpp
clang_staticanalyzer_warnings_01.cpp:3:10: warning: Value stored to 'x' during
its initialization is never read
    int *x = new int;
         ^   ~~~~~~~
clang_staticanalyzer_warnings_01.cpp:3:10: note: Value stored to 'x' during its
initialization is never read
    int *x = new int;
         ^   ~~~~~~~
clang_staticanalyzer_warnings_01.cpp:4:1: warning: Potential leak of memory
pointed to by 'x'
}
^
clang_staticanalyzer_warnings_01.cpp:3:14: note: Memory is allocated
    int *x = new int;
             ^~~~~~
(...)
2 warnings generated.
```

# Usage on single translation units (cont'd)

Run on single file (with *html* output):

```
$ clang++ --analyze -Xanalyzer -analyzer-output=html clang_sa_warnings_01.cpp
```

```
1  int main()
2  {
3      int *x = new int;
4  }
```

1 Memory is allocated →

2 ← Potential leak of memory pointed to by 'x'

```
1  int main()
2  {
3      int *x = new int;
4  }
```

Value stored to 'x' during its initialization is never read

# Usage on whole projects

For whole projects, *scan-build* is required:

**Running scan-build**

```
$ → cd to build dir
$ → clean build dir
$ scan-build cmake /path/to/source
$ scan-build make
... now compiles and analyzes source code ...
```

- How does it work?
  - *scan-build* overrides the *CC* and *CXX* variables to use a fake compiler instead
  - fake compiler executes original compiler and then the static analyzer on the translation unit
  - the output of *scan-build* is a set of HTML files, viewable in browser

# Usage on whole projects -- viewing results

**Viewing results**

```
$ ...
$ scan-build make
...
scan-build: Run 'scan-view /tmp/scan-build-XYZ' to examine bug reports.
$ scan-view /tmp/scan-build-XYZ
```

scan-view will open a web browser in this format:

**Bug Summary**

| Bug Type | Quantity | Display? |
|---|---|---|
| **All Bugs** | **2** | ☑ |
| **Dead store** | | |
| Dead initialization | 1 | ☑ |
| **Memory error** | | |
| Memory leak | 1 | ☑ |

**Reports**

| Bug Group | Bug Type ▾ | File | Function/Method | Line | Path Length | | | |
|---|---|---|---|---|---|---|---|---|
| Dead store | Dead initialization | clang_sa_warnings_01.cpp | main | 3 | 1 | View Report | Report Bug | Open File |
| Memory error | Memory leak | clang_sa_warnings_01.cpp | main | 4 | 2 | View Report | Report Bug | Open File |

# Clang Static Analyzer: Extra

- Most useful scan-build options:
  - *-h* to list available options and checkers
  - *-enable-checker [checker_name]* to enable individual checkers
  - *-o* to specifiy the HTML output dir
- Windows users: Use *scan-build.bat* instead

Demo: Viewing *scan-build* results in browser

- LLVM/Clang Overview
- Clang compiler frontend (static analysis)
- Clang Static Analyzer (static analysis)
- **clang-tidy (static analysis / linter / refactoring)**
- Clazy (static analysis, Qt centric)
- clang-format (coding style)
- Clang Sanitizers (dynamic analysis)

# clang-tidy

- Detect bug prone coding patterns
- Enforce coding conventions
- Advocate modern and maintainable code
- Checks can be more expensive than compilation
  - Depends on how involved the checker is
- Easy to extend with custom checks
  - E.g. to enfore company code policies

# But, wait...

Different tools for static analysis (Clang, Clang SA, clang-tidy), what the...?!

- Explanation from the developers:
  - **Clang diagnostic**:
    - if the check is generic enough
    - targets code patterns that most probably are bugs (rather than style or readability issues)
    - can be implemented efficiently and with extremely low false positive rate
  - **Clang static analyzer check**:
    - if the check requires some sort of control flow analysis
  - **clang-tidy check**:
    - is a good choice for linter-style checks
    - checks that are related to a certain coding style
    - checks that address code readability, etc.

# clang-tidy: Intro to checks

- Has several built-in checks
  - List them with `clang-tidy --list-checks -checks='*'`
- Can run checks from Clang Static Analyzer(!)
- Extensible by writing **custom checks**
  - Checks are organized in modules
  - Can be linked into clang-tidy
  - Also see: http://clang.llvm.org/extra/clang-tidy/Contributing.html

# clang-tidy: Checks categories

- *boost*-*: Checks related to Boost library.
- *cert*-*: Checks related to CERT Secure Coding Guidelines.
- *cppcoreguidelines*-*: Checks related to C++ Core Guidelines.
- *clang-analyzer*-*: Clang Static Analyzer checks.
- *google*-*: Checks related to the Google coding conventions.
- *llvm*-*: Checks related to the LLVM coding conventions.
- *misc*-*: Checks that we didn't have a better category for.
- *modernize*-*: Checks that advocate usage of modern language constructs.
- *mpi*-*: Checks related to MPI (Message Passing Interface).
- *performance*-*: Checks that target performance-related issues.
- *readability*-*: Checks that target readability-related issues that don't relate to any particular coding style.

# clang-tidy checks: Some useful checks

- *google-explicit-constructor*
- *performance-unnecessary-value-param*
- *readability-inconsistent-declaration-parameter-name*
- *hicpp-explicit-conversions*
- *hicpp-use-equals-default*
- *hicpp-use-equals-delete*
- *hicpp-undelegated-constructor*
- *modernize-** (most of them are useful!)

# clang-tidy: Check one file

# clang-tidy: Check one file

Example Code (clang_tidy_warnings_01.cpp)

```
 1 enum Things { Dog, Frog };
 2
 3 int bar(Things thing)
 4 {
 5     int ret;
 6     switch (thing) {
 7         case Dog: case Frog: ret = 0;
 8         default: ret = -1;
 9     }
10     return ret;
11 }
```

# clang-tidy: Check one file

Example Code (clang_tidy_warnings_01.cpp)

```
 1 enum Things { Dog, Frog };
 2
 3 int bar(Things thing)
 4 {
 5     int ret;
 6     switch (thing) {
 7         case Dog: case Frog: ret = 0;
 8         default: ret = -1;
 9     }
10     return ret;
11 }
```

```
$ clang-tidy clang_tidy_warnings_01.cpp --
1 warning generated.
clang_tidy_warnings_01.cpp:7:30: warning: Value stored to 'ret' is never read
[clang-analyzer-deadcode.DeadStores]
        case Dog: case Frog: ret = 0;
                             ^
clang_tidy_warnings_01.cpp:7:30: note: Value stored to 'ret' is never read
```

# clang-tidy: Check whole project

- Prerequisite: Needs a *Compilation Database* in build dir
- Note that clang-tidy is *single-threaded*!
- Needs wrapper script to be run in parallel: run-clang-tidy.py

**Running run-clang-tidy on example project**

```
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
$ run-clang-tidy.py
Enabled checks:
    clang-analyzer-core.CallAndMessage
    clang-analyzer-core.DivideZero
    ...

... runs clang-tidy on each translation unit ...
clang-tidy-8 ... .../clang_sa_warnings_01.cpp
clang_sa_warnings_01.cpp:3:10: warning: Value stored to 'x' during its
initialization is never read [clang-analyzer-deadcode.DeadStores]
    int *x = new int;
         ^
...
2 warnings generated.

clang-tidy-8 ... .../clang_warnings_02.cpp
...
```

# clang-tidy: Refactor one file

- To refactor, run clang-tidy with *-fix* option!

# clang-tidy: Refactor one file

- To refactor, run clang-tidy with *-fix* option!

# clang-tidy: Refactor one file

- To refactor, run clang-tidy with *-fix* option!

Example Code (clang_tidy_modernize_01.cpp)

```
 1 struct Base
 2 {
 3     virtual ~Base() {}
 4     virtual void foo() = 0;
 5 };
 6
 7 struct Derived : public Base
 8 {
 9     virtual void foo() {}
10 };
```

# clang-tidy: Refactor one file

- To refactor, run clang-tidy with *-fix* option!

Example Code (clang_tidy_modernize_01.cpp)

```
 1 struct Base
 2 {
 3     virtual ~Base() {}
 4     virtual void foo() = 0;
 5 };
 6
 7 struct Derived : public Base
 8 {
 9     virtual void foo() {}
10 };
```

Clang-tidy run

```
$ clang-tidy -checks='-*,modernize-use-override' clang_tidy_modernize_01.cpp -fix
-- -std=c++11
```

# clang-tidy: Refactor one file

- To refactor, run clang-tidy with *-fix* option!

**Example Code (clang_tidy_modernize_01.cpp)**

```
 1 struct Base
 2 {
 3     virtual ~Base() {}
 4     virtual void foo() = 0;
 5 };
 6
 7 struct Derived : public Base
 8 {
 9     virtual void foo() {}
10 };
```

**Clang-tidy run**

```
$ clang-tidy -checks='-*,modernize-use-override' clang_tidy_modernize_01.cpp -fix
-- -std=c++11
```

**Results in Patch**

```
@@ -6,5 +6,5 @@ struct Base

 struct Derived : public Base
 {
-    virtual void foo() {}
+    void foo() override {}
 };
```

# clang-tidy: Refactor whole project

- Prerequisite: Needs a *Compilation Database* in build dir

```
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..

$ run-clang-tidy.py -header-filter='.*' -checks='-*,modernize-use-override' -fix
Enabled checks:
    modernize-use-override
    ...


...
clang-tidy-8 ... .../clang_tidy_modernize_01.cpp
.../clang_tidy_modernize_01.cpp:9:18: warning: prefer using 'override' or
(rarely) 'final' instead of 'virtual' [modernize-use-override]
    virtual void foo() {}
    ~~~~~~~~     ^
                        override

1 warning generated.
```

**Important note**
run-clang-tidy.py will only *apply* changes at the end of the full run!

# JSON Compilation Database

- A file called *compile_commands.json*
- Contains information how to translation units are being compiled

**Example JSON Compilation Database**

```
{
  "directory": ".../build/icemon",
  "command": "/usr/bin/c++  -DQT_CORE_LIB -I... .../fakemonitor.cc"
}
{
  "directory": ".../build/icemon",
  "command": "/usr/bin/c++  -DQT_CORE_LIB -I... .../hostinfo.cc"
}
...
```

# Compilation Database: HOWTO create?

Build system specific, some build systems can generate them

- Very easy with *CMake* : via `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
- For build systems which don't generate one:
  - Make use of Bear (Unix only!)
    - Simply run: `bear make`
  - Last resort: Write a script yourself!
- Also see: Series of blog posts: https://www.kdab.com/tag/clang/

Demo: Refactoring whole project with clang-tidy

# Extending clang-tidy

- Not covered in this slide deck
  - Excellent blog series by Stephen Kelly:
    - https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-1-extending-clang-tidy/
    - https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/
    - https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-3-rewriting-code-with-clang-tidy/

- LLVM/Clang Overview
- Clang compiler frontend (static analysis)
- Clang Static Analyzer (static analysis)
- clang-tidy (static analysis / linter / refactoring)
- **Clazy (static analysis, Qt centric)**
- clang-format (coding style)
- Clang Sanitizers (dynamic analysis)

# Static Code Analysis: Clazy

- A Clang plugin
- Finds common bugs and performance pitfalls in Qt and C++ code
- Intro: www.kdab.com/use-static-analysis-improve-performance
- Project homepage: https://github.com/KDE/clazy
- Cross-platform, currently supported are macOS, MSVC2015+ and Linux
- *LGPLv2* licensed, a KDE project
- Integrated in and shipped with *Qt Creator* IDE

# Clazy: Examples

Example Code (1)

```
1 #ifndef GL_FRAMEBUFFER_SRB
2 #define GL_FRAMEBUFFER_SRGB 0x8DB9
3 #endif
```

# Clazy: Examples

**Example Code (1)**

```
1 #ifndef GL_FRAMEBUFFER_SRB
2 #define GL_FRAMEBUFFER_SRGB 0x8DB9
3 #endif
```

**Clazy Run (1)**

```
qplatformbackingstore.cpp:4:9: warning: Possible typo in define.
GL_FRAMEBUFFER_SRB vs GL_FRAMEBUFFER_SRGB [-Wclazy-ifndef-define-typo]
```

# Clazy: Examples

**Example Code (1)**

```
1 #ifndef GL_FRAMEBUFFER_SRB
2 #define GL_FRAMEBUFFER_SRGB 0x8DB9
3 #endif
```

**Clazy Run (1)**

```
qplatformbackingstore.cpp:4:9: warning: Possible typo in define.
GL_FRAMEBUFFER_SRB vs GL_FRAMEBUFFER_SRGB [-Wclazy-ifndef-define-typo]
```

**Example code (2)**

```
1 class Base
2 {
3 public:
4     virtual ~Base() = default;
5 };
```

# Clazy: Examples

**Example Code (1)**

```
1 #ifndef GL_FRAMEBUFFER_SRB
2 #define GL_FRAMEBUFFER_SRGB 0x8DB9
3 #endif
```

**Clazy Run (1)**

```
qplatformbackingstore.cpp:4:9: warning: Possible typo in define.
GL_FRAMEBUFFER_SRB vs GL_FRAMEBUFFER_SRGB [-Wclazy-ifndef-define-typo]
```

**Example code (2)**

```
1 class Base
2 {
3 public:
4     virtual ~Base() = default;
5 };
```

**Clazy Run (2)**

```
main.cpp:36:1: warning: Polymorphic class Base is copyable. Potential slicing.
[-Wclazy-copyable-polymorphic]
```

# Clazy: Examples (cont'd)

**Example Code**

```
1 void test(QObject *obj)
2 {
3     int a = 1;
4     auto f = [&a]() { /* ... */ };
5     obj->connect(obj, &QObject::destroyed, [a]() { /* ... */ });
6     obj->connect(obj, &QObject::destroyed, [&a]() { /* ... */ });
7 }
```

**Clazy Run**

```
$ clazy -isystem /usr/include/qt/ -std=c++11 tests/lambda-in-connect/main.cpp
```

# Clazy: Examples (cont'd)

**Example Code**

```
1 void test(QObject *obj)
2 {
3     int a = 1;
4     auto f = [&a]() { /* ... */ };
5     obj->connect(obj, &QObject::destroyed, [a]() { /* ... */ });
6     obj->connect(obj, &QObject::destroyed, [&a]() { /* ... */ });
7 }
```

**Clazy Run**

```
$ clazy -isystem /usr/include/qt/ -std=c++11 tests/lambda-in-connect/main.cpp
```

```
1 tests/lambda-in-connect/main.cpp:15:46: warning: capturing local variable
2 by reference in lambda [-Wclazy-lambda-in-connect]
3     obj->connect(obj, &QObject::destroyed, [&a]() {});
4                                              ^
```

# Clazy: Examples (cont'd)

**Example Code (clazy_qstring_allocations.cpp)**

```
 1 #include <QtCore/QString>
 2
 3 bool isFoo(const QString &string)
 4 {
 5     return string == "foo";
 6 }
 7
 8 QString foo()
 9 {
10     return "foo";
11 }
```

**Clazy Run**

```
$ export CLAZY_CHECKS=level0,level1,level2
$ clazy -isystem ... -std=c++11 clazy_qstring_allocations.cpp
```

# Clazy: Examples (cont'd)

**Example Code (clazy_qstring_allocations.cpp)**

```
 1 #include <QtCore/QString>
 2
 3 bool isFoo(const QString &string)
 4 {
 5     return string == "foo";
 6 }
 7
 8 QString foo()
 9 {
10     return "foo";
11 }
```

**Clazy Run**

```
$ export CLAZY_CHECKS=level0,level1,level2
$ clazy -isystem ... -std=c++11 clazy_qstring_allocations.cpp
```

```
1 clazy_qstring_allocations.cpp:5:12: warning: QString(const char*) being called
2 [-Wclazy-qstring-allocations]
3 clazy_qstring_allocations.cpp:10:12: warning: QString(const char*) being called
4 [-Wclazy-qstring-allocations]
```

# Clazy: Examples using Fixits

- Some errors can be fixed automatically
- Only do this to code tracked by a VCS

```
$ clazy -isystem /usr/include/x86_64-linux-gnu/qt5 -std=c++11 -fPIC
  -Xclang -plugin-arg-clazy -Xclang qstring-allocations
  -Xclang -plugin-arg-clazy -Xclang export-fixes
  -c clazy_qstring_allocations.cpp
... generates a .yaml file in the current dir ...
$ clang-apply-replacements .
$ git diff
```

```
@@ -2,10 +2,10 @@

 bool isFoo(const QString &string)
 {
-    return string == "foo";
+    return string == QLatin1String("foo");
 }

 QString foo()
 {
-    return "foo";
+    return QStringLiteral("foo");
 }
```

# Clazy: Running it on a project

- The *clazy* executable needs to be aware of command-line arguments
  - Not feasible to do this manually for a whole project
- The *clazy-standalone* executable infers that info from a *JSON Compilation database*
  - Just like other Clang tools, such as *clang-tidy*
- Examples:

**Running Clazy on all .cpp files in a project**

```
$ find . -name "*.cpp" | xargs
clazy-standalone -checks=level2 -p $BUILD_DIR
```

**Tip: Running Clazy on all files in a project using *jq***

```
$ jq ".[] | .file" $BUILD_DIR/compile_commands.json | xargs
clazy-standalone -checks=level2 -p $BUILD_DIR
```

- LLVM/Clang Overview
- Clang compiler frontend (static analysis)
- Clang Static Analyzer (static analysis)
- clang-tidy (static analysis / linter / refactoring)
- Clazy (static analysis, Qt centric)
- **clang-format (coding style)**
- Clang Sanitizers (dynamic analysis)

- Motivation
  - Developers waste time on formatting
  - Consistent coding style is important
- clang-format provides
  - Automatic formatting
  - Supports different style guides
  - Fine-tuning of formatting rules

# clang-format (cont'd)

- Similar to most other code formatters
- Can be run on individual files or subdirs
- Parametrization via command-line or config file
- Less flexible with regards to parameterization as other formatters
  - for instance *uncrustify* is extremely configurable
- Offers script for patch reformatting (*clang-format-diff.py*)

# clang-format: Examples

# clang-format: Examples

Example Code (clang_format_example.cpp)

```
1 int main()
2 {
3 int i=0;
4 }
```

# clang-format: Examples

**Example Code (clang_format_example.cpp)**

```
1 int main()
2 {
3 int i=0;
4 }
```

**clang-format run**

```
clang-format -style='Mozilla' -i clang_format_example.cpp
```

# clang-format: Examples

**Example Code (clang_format_example.cpp)**

```
1 int main()
2 {
3 int i=0;
4 }
```

**clang-format run**

```
clang-format -style='Mozilla' -i clang_format_example.cpp
```

**Results in**

```
1 @@ -1,4 +1,5 @@
2 -int main()
3 +int
4 +main()
5  {
6 -int i=0;
7 +  int i = 0;
8  }
```

- LLVM/Clang Overview
- Clang compiler frontend (static analysis)
- Clang Static Analyzer (static analysis)
- clang-tidy (static analysis / linter / refactoring)
- Clazy (static analysis, Qt centric)
- clang-format (coding style)
- **Clang Sanitizers (dynamic analysis)**

# Sanitizers

- Fast error checkers supported by GCC 4.9+ and Clang 3.1+
- Successfully applied to large applications, e.g. Google Chrome, Firefox...
- Available sanitizers:
  - `address`: memory error detector
  - `thread`: detect data races
  - `undefined`: check code for undefined behavior
  - And more, with mixed maturity level

Sanitizers are compiler addons:

- They insert checks or modify code to spot common programming errors.
- These checks come with a significant CPU and memory overhead.
  - But, it is much lower than Valgrind's virtual machine.
- Threaded code is not artificially serialized and can execute in parallel.
- Read the ASan paper for more information on the internals:

  Serebryany, Konstantin, et al. "AddressSanitizer: A Fast Address Sanity Checker." USENIX Annual Technical Conference. 2012. APA
  www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

# Enabling Sanitizers

- Compile your code with `-fsanitize={address,leak,undefined,thread}`.
- It's also useful to apply this to your dependencies, e.g. Qt.
  - Otherwise, some errors cannot be detected.
- Enabling sanitizers for Qt globally:
  - Configure with `-sanitize {address,leak,undefined,thread}`.
  - All code compiled against this Qt will inherit the sanitizer settings.
- Enabling sanitizers in individual QMake projects:
  - `qmake -r CONFIG+="sanitizer sanitize_address" ...`
- Enabling sanitizers in CMake projects:
  - Leverage the Extra CMake Modules (ECM):
    api.kde.org/ecm/module/ECMEnableSanitizers.html
  - Make sanitizers available in your `CMakeLists.txt` with:
    `include(ECMEnableSanitizers)`
  - Enable sanitizers via:
    `cmake -DECM_ENABLE_SANITIZERS="address;undefined" ...`

# Sanitizers on Windows

- Requires Clang for Windows.
  - Windows 10 Fall Creator not supported by Clang 5.
  - Wait for Clang 6 or compile Clang from sources.
    - Ensure compiler-rt uses the dynamic runtime (`/MD`)
- Sanitizers do not work with debug runtimes
- Sanitizer reports show up in DbgView
- Create a custom `mkspec` called `win32-clang-msvc-sanitizers`
  - Copy the `qplatformdefs.h` from `win32-clang-msvc`.
  - Create a custom `qmake.conf`:

```
1  include(../win32-clang-msvc/qmake.conf)
2
3  CONFIG += force_debug_info
4  QMAKE_CXXFLAGS_RELEASE_WITH_DEBUGINFO += -fsanitize=address,undefined
5  # optionally: disable compiler optimizations
6  QMAKE_CXXFLAGS_RELEASE_WITH_DEBUGINFO -= -O2
7  QMAKE_CXXFLAGS_RELEASE_WITH_DEBUGINFO += -Og
8
9  # link against the ASAN runtime, the path below is the compiler-rt install prefix
10 LIBS += /LIBPATH:C:/Qt/src/compiled/clang/lib/clang/6.0.0/lib/windows/
11 LIBS += -lclang_rt.asan_dynamic-x86_64 -lclang_rt.asan_dynamic_runtime_thunk-x86_64
```

- Use it with QMake: `qmake -spec win32-clang-msvc-sanitizers`

# Address Sanitizer (ASan)

- Use `-fsanitize=address` to detect:
  - Out-of-bounds accesses to heap, stack and globals
  - Use-after-free
  - Use-after-return (to some extent)
  - Double-free, invalid free
  - ~~Memory leaks~~ (Not supported on Windows)
- Passing `-fsanitize=leak` only activates the low-overhead leak checking
- **Note:** Application will terminate when the first error is detected
- Documentation:
  - github.com/google/sanitizers/wiki/AddressSanitizer
  - clang.llvm.org/docs/AddressSanitizer.html

# Address Sanitizer (ASan) - Example

```
1 int *i = new int[2];
2 int read = i[2];
3 i[3] = 1; // and write
```

**Demo: debugging/ex_invalid_readwrite**

# Address Sanitizer (ASan) - Example

```
1 int *i = new int[2];
2 int read = i[2];
3 i[3] = 1; // and write
```

Running the example compiled with `-fsanitize=address` yields:

```
1  ==32417== ERROR: AddressSanitizer: heap-buffer-overflow
2      on address 0x60200000de98 at pc 0x000000400857 bp 0x7ffd6ee14f40 sp 0x7ffd6ee14f30
3  READ of size 4 at 0x60200000de98 thread T0
4      #0 0x400856 in main .../ex_invalid_readwrite/ex_invalid_readwrite.cpp:4
5
6  0x60200000de98 is located 0 bytes to the right of 8-byte region
7      [0x60200000de90,0x60200000de98)
8  allocated by thread T0 here:
9      #0 0x7fd8761a3a62 in operator new[](unsigned long) .../asan_new_delete.cc:62
10     #1 0x400817 in main .../debugging/ex_invalid_readwrite/ex_invalid_readwrite.cpp:3
```

If no symbols are shown, try exporting

```
ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer
ASAN_OPTIONS=symbolize=1
```

**Demo: debugging/ex_invalid_readwrite**

# Address Sanitizer (ASan) - Error Recovery

- Execution is terminated after the first error.
- This is just default behavior.
- Enable error recovery:
  - Compile with `-fsanitize-recover=all`.
  - Set environment variable `ASAN_OPTIONS=halt_on_error=0`.

```
==32417== ERROR: AddressSanitizer: heap-buffer-overflow
    on address 0x60200000de98 at pc 0x000000400857 bp 0x7ffd6ee14f40 sp 0x7ffd6ee14f30
READ of size 4 at 0x60200000de98 thread T0
    #0 0x400856 in main .../ex_invalid_readwrite/ex_invalid_readwrite.cpp:4

...

==13213==ERROR: AddressSanitizer: heap-buffer-overflow
    on address 0x60200000efdc at pc 0x000000400900 bp 0x7fffd0a77a30 sp 0x7fffd0a77a20
WRITE of size 4 at 0x60200000efdc thread T0
    #0 0x4008ff in main .../ex_invalid_readwrite/ex_invalid_readwrite.cpp:10
```

Note: Errors after the first one *may* be spurious.

**Demo: debugging/ex_invalid_readwrite**

# Leak Sanitizer (LSan)

- Low-overhead memory leak detector
- Documentation:
  - github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

# Leak Sanitizer (LSan)

- Low-overhead memory leak detector
- Documentation:
  - github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

```
1 ==7668==ERROR: LeakSanitizer: detected memory leaks
2
3 Direct leak of 100 byte(s) in 1 object(s) allocated from:
4     #0 0x7fd8c003fb18 in operator new[](unsigned long) /build/gcc/src/gcc/libsanitizer/as
5     #1 0x558d36621f3b in foo() ../../ex_leak/ex_leak.cpp:9
6     #2 0x558d3662203e in main ../../ex_leak/ex_leak.cpp:16
7     #3 0x7fd8bf0dfee2 in __libc_start_main (/usr/lib/libc.so.6+0x26ee2)
```

# Leak Sanitizer (LSan)

- Low-overhead memory leak detector
- Documentation:
  - github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

```
1 ==7668==ERROR: LeakSanitizer: detected memory leaks
2
3 Direct leak of 100 byte(s) in 1 object(s) allocated from:
4     #0 0x7fd8c003fb18 in operator new[](unsigned long) /build/gcc/src/gcc/libsanitizer/as
5     #1 0x558d36621f3b in foo() ../../ex_leak/ex_leak.cpp:9
6     #2 0x558d3662203e in main ../../ex_leak/ex_leak.cpp:16
7     #3 0x7fd8bf0dfee2 in __libc_start_main (/usr/lib/libc.so.6+0x26ee2)
```

- Useful to detect leaks in unit tests
  - Exit code will indicate failure when leaks are detected
- Suppression files can be used to ignore third-party or system libraries
  - `export LSAN_OPTIONS=suppressions=/absolute/path/to/file`
- Backtraces can break when encountering third party libraries
  - The default unwinder relies on `-fno-omit-frame-pointers`
  - Workaround this via `LSAN_OPTIONS=fast_unwind_on_malloc=true`

# Undefined Behavior Sanitizer (ubsan)

- Use `-fsanitize=undefined` to detect undefined behavior, such as:
  - Invalid bit shifting
  - Integer overflow
  - Null pointer dereferencing
  - Out-of-bounds access on stack arrays
  - Pointer alignment
  - And more
- Can be combined with ASan: `-fsanitize=address,undefined`
- Does not terminate when an error is found
- Background information:
  developerblog.redhat.com/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan

# Undefined Behavior Sanitizer (ubsan) - Example

Assuming `argc = 1`, where is undefined behavior in the following code?

```
1 int foo = 23 << (argc * 32);
2
3 int bar = std::numeric_limits<int>::min() * argc;
4 unsigned int asdf = -bar;
```

**Demo: debugging/ex_undefined_behavior**

# Undefined Behavior Sanitizer (ubsan) - Example

Assuming `argc = 1`, where is undefined behavior in the following code?

```
1 int foo = 23 << (argc * 32);
2
3 int bar = std::numeric_limits<int>::min() * argc;
4 unsigned int asdf = -bar;
```

Running the example compiled with `-fsanitize=undefined` yields:

```
1 main.cpp:7:11: runtime error: shift exponent 32 is too large for 32-bit type 'int'
2
3 main.cpp:12:18: runtime error: negation of -2147483648 cannot be represented
4     in type 'int'; cast to an unsigned type to negate this value to itself
```

Hint: Set `UBSAN_OPTIONS=print_stacktrace=1`

**Demo: debugging/ex_undefined_behavior**

# Lab: Sanitizers

- Compile and run the lab.
- It crashes instantly.
- Use the sanitizers to find the bug.
- Fix the bug and repeat the steps.
- Finally, try setting a birthday in 1750, and check the age in seconds.

**Lab: debugging/lab_dialog**

# Thread Sanitizer (TSan)

- Use `-fsanitize=thread` to detect:
  - Data races
  - Thread leaks
  - Destruction of locked mutex
  - Deadlocks
- Documentation:
  - https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
  - http://clang.llvm.org/docs/ThreadSanitizer.html

**Note:** Only supported on 64bit x86 machines running Linux.

# Thread Sanitizer (TSan) - Preparing Qt

Custom Qt build with Thread Sanitizer support is necessary for proper interception of `QMutex` et al.:

```
configure \
    -platform linux-clang-libc++ \
    -sanitize thread \
    -debug ...
```

- GCC and `libstdc++` should also work in principle.
  - In practice, Clang's TSan with `libc++` is more reliable.
- Alternatively, replace `-debug` with `-release -force-debug-info`.

# Thread Sanitizer (TSan) - Example

```cpp
void race(int *counter, int increment) { *counter += increment; }

int main()
{
    int counter = 0;
    {
        std::vector<std::future<void>> futures;
        const auto threadsToStart = std::thread::hardware_concurrency();
        for (unsigned i = 0; i < threadsToStart; ++i)
            futures.push_back(std::async(std::launch::async, race, &counter, i));
    }
    std::cout << counter << std::endl;
    return 0;
}
```

```
WARNING: ThreadSanitizer: data race (pid=4563)
    Read of size 4 at 0x7ffeb2e34adc by thread T2:
        #0 race(int*, int) ex_datarace/main.cpp:1 (main+0x0000004017ad)
        ...
    Previous write of size 4 at 0x7ffeb2e34adc by thread T1:
        #0 race(int*, int) ex_datarace/main.cpp:1 (main+0x0000004017c5)
        ...
```

**Demo: debugging/ex_datarace**

# Sanitizers vs. Valgrind

| | Sanitizer | Valgrind |
|---|---|---|
| **heap buffer overflow** | Address | Memcheck |
| **dangling pointer** | Address | Memcheck |
| **mismatched new/delete** | Address | Memcheck |
| **memory leak** | Address, Leak | Memcheck |
| **uninitialized memory** | Memory (WIP) | Memcheck |
| **stack buffer overflow** | Address | SGCheck (experimental) |
| **integer overflow** | Undefined | - |
| **alignment fault** | Undefined | - |
| **data race** | Thread | Helgrind, DRD |
| **deadlock** | Thread | Helgrind, DRD |

- Sanitizers are much faster and use less memory.
- No recompilation is required for Valgrind.
  - All libraries linked into your application will be checked by Valgrind.

# Summary

- Clang compiler
  - Useful static analysis at high-level warning level **during normal compilation**
- Static analyzer
  - Deep control flow analysis, slow
  - HTML-formatted overview over issues
- clang-tidy (depends on CMake compdb)
  - C++ linter,
  - Ability to fix code automatically
- Clazy
  - Clang plugin providing Qt/C++ specific diagnostics
  - Similar usage possible like for clang-tidy
- clang-format
  - Simple-to-use code formatter, limited flexibility
- Clang Sanitizers
  - Dynamic analysis tools
  - Detecting erratic behavior at runtime

# Installation

- Linux: Grab it via your package manager
- Windows: Get via LLVM website
- macOS: Homebrew: `$ brew install --with-toolchain llvm` (or ?)
- Downloads for basically all platforms:
  - http://releases.llvm.org/download.html

# Even more tools

- Not discussed, but also useful:
  - Include-what-you-use
    - Project website: https://include-what-you-use.org/
    - Strips unnecessary includes from source files
    - Verdict: Nice in theory, but breaks when using forwarding headers (hey Qt!)
  - clang-query
    - Use Clang AST matchers to query code for symbols
    - Good intro: https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools
    - Verdict: Nice tool, but also super slow for too-broad queries. Tool may freeze.

**clang-query example**

```
clang-query> match ifStmt(hasCondition(binaryOperator(hasOperatorName("==")).bind("op")))

Match #1:

/tmp/iflhsptr.c:2:7: note: "op" binds here
  if (p == 0) {
      ^~~~~~
```

# Links

- Entry point: http://clang.llvm.org/
  - Important for setting up your project (→ compdb, etc.)
    - http://clang.llvm.org/docs/HowToSetupToolingForLLVM.html
  - Clang Static Analyzer:
    - http://clang-analyzer.llvm.org/scan-build.html / http://clang-analyzer.llvm.org/
  - clang-tidy:
    - http://clang.llvm.org/extra/clang-tidy.html
  - Clazy:
    - https://github.com/KDE/clazy
  - clang-format:
    - http://clang.llvm.org/docs/ClangFormat.html

Questions?

www.kdab.com
kevin.funk@kdab.com

KDAB is a provider for expert Qt, OpenGL and C++ services