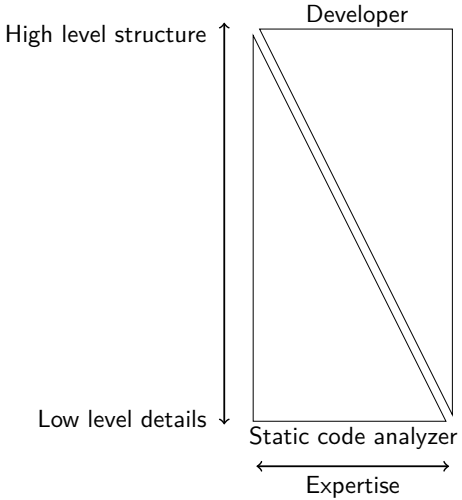
Things I Learned From The Static Analyzer

Bart Verhagen
bart@verhagenconsultancy.be

15 November 2019

What can they do for us?



Focus areas

- Consistency
- Maintenance
- Performance
- Prevent bugs and undefined behaviour

Used static code analyzers for this talk

- clang-tidy
- cppcheck

???

```
#include "someType.h"

class Type;
template<typename T> class TemplateType;

typedef TemplateType<Type> NewType;
```

Typedef vs using

```cpp
#include "someType.h"

class Type;
template<typename T> class TemplateType;

typedef TemplateType<Type> NewType;
using NewType = TemplateType<Type>;

// typedef TemplateType<T> NewTemplateType; // Compilation error

template<typename T>
using NewTemplateType = TemplateType<T>;
```

modernize-use-using

Warning: use 'using' instead of 'typedef'

The Clang Team. *Clang-tidy - modernize-use-using*. URL: https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-using.html

???

```
class Class {
    public:
        Class() : m_char('1'),
            m_constInt(2) {}

        explicit Class(char some_char) :
            m_char(some_char),
            m_constInt(2) {}

        explicit Class(int some_const_int) :
            m_char('1'),
            m_constInt(some_const_int) {}

        Class(char some_char, int some_const_int) :
            m_char(some_char),
            m_constInt(some_const_int) {}

    private:
        char m_char;
        const int m_constInt;
};
```

Default member initialization

```
class Class {
    public:
        Class() : m_char('1'),
            m_constInt(2) {}

        explicit Class(char some_char) :
            m_char(some_char),
            m_constInt(2) {}

        explicit Class(int some_const_int) :
            m_char('1'),
            m_constInt(some_const_int) {}

        Class(char some_char, int some_const_int) :
            m_char(some_char),
            m_constInt(some_const_int) {}

    private:
        char m_char;
        const int m_constInt;
};
```

modernize-use-default-member-init

Warning: use default member initializer for 'm_char'
Warning: use default member initializer for 'm_constInt'

The Clang Team. *Clang-tidy - modernize-use-default-member-init*. URL: https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-default-member-init.html

cppreference.com. *direct initialization*. URL: https://en.cppreference.com/w/cpp/language/direct_initialization

Default member initialization

```
class Class {
    public:
        explicit Class(char some_char) :
            m_char(some_char) {}

        explicit Class(int some_const_int) :
            m_constInt(some_const_int) {}

        Class(char some_char, int some_const_int) :
            m_char(some_char),
            m_constInt(some_const_int) {}

    private:
        char m_char = '1';
        const int m_constInt = {2};
};
```

Delegating constructors

```
class Class {
    public:
        explicit Class(char some_char) : Class(some_char, 2) {}

        explicit Class(int some_const_int) : Class('1', some_const_int) {}

        Class(char some_char, int some_const_int) :
            m_char(some_char),
            m_constInt(some_const_int) {}

    private:
        char m_char = '1';
        const int m_constInt = {2};
};
```

cppreference.com. *Delegating constructor*. URL: https://en.cppreference.com/w/cpp/language/initializer_list#Delegating_constructor

???

```cpp
#include <cstdlib>
#include <random>
#include <vector>

auto getRandomSeries(std::size_t size, double min, double max) -> std::vector<double> {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(min, max);

    std::vector<double> result;
    for(std::size_t i = 0U; i < size; ++i) {
        result.emplace_back(dis(gen));
    }
    return result;
};

auto main() -> int {
    constexpr auto threshold = 5.5;
    auto series = getRandomSeries(1024U, 0.0, 10.0);
    for(const auto& value : series) {
        if(value < threshold) {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

Use <algorithm>

```cpp
#include <cstdlib>
#include <random>
#include <vector>

auto getRandomSeries(std::size_t size, double min, double max) -> std::vector<double> {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(min, max);

    std::vector<double> result;
    for(std::size_t i = 0U; i < size; ++i) {
        result.emplace_back(dis(gen));
    }
    return result;
};

auto main() -> int {
    constexpr auto threshold = 5.5;
    auto series = getRandomSeries(1024U, 0.0, 10.0);
    for(const auto& value : series) {
        if(value < threshold) {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

useStlAlgorithm

Consider using std::any_of algorithm instead of a raw loop.

Use <algorithm>

```cpp
#include <algorithm>
#include <cstdlib>
#include <random>
#include <vector>

namespace  {
    auto getRandomSeries(size_t size, double min, double max) -> std::vector<double> {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<> dis(min, max);

        std::vector<double> result;
        std::generate_n(std::back_inserter(result), size, [&dis, &gen]() { return dis(gen); });
        return result;
    };
} // namespace

auto main() -> int {
    constexpr auto threshold = 5.5;
    auto series = getRandomSeries(1024U, 0.0, 10.0);

    if(std::any_of(series.begin(), series.end(), [threshold](auto value) { return value >= threshold; })) {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

???

```cpp
#include <fstream>

class FileResource {
    public:
        FileResource() {}
        explicit FileResource(const std::string& name) {
            m_output.open(name, std::ios::out);
        }
        FileResource(FileResource&& other) noexcept :
            m_output(std::move(other.m_output)) {}
        ~FileResource() {}

        auto operator=(FileResource&& other) noexcept -> FileResource& {
            m_output = std::move(other.m_output);
            return *this;
        }
    private:
        FileResource(const FileResource& other);    // Declaration only
        auto operator=(const FileResource& other) -> FileResource&; // Declaration only

        std::ofstream m_output;
};
```

Defaulting/deleting constructors/destructors

```cpp
#include <fstream>

class FileResource {
    public:
        FileResource() {}
        explicit FileResource(const std::string& name) {
            m_output.open(name, std::ios::out);
        }
        FileResource(FileResource&& other) noexcept :
            m_output(std::move(other.m_output)) {}
        ~FileResource() {}

        auto operator=(FileResource&& other) noexcept -> FileResource& {
            m_output = std::move(other.m_output);
            return *this;
        }
    private:
        FileResource(const FileResource& other);    // Declaration only
        auto operator=(const FileResource& other) -> FileResource&; // Declaration only

        std::ofstream m_output;
};
```

hicpp-use-equals-default

Warning: use '= default' to define a trivial default constructor/destructor
Warning: use '= delete' to prohibit calling of a special member function

The Clang Team. *Clang-tidy - hicpp-use-equals-default*. URL: https://clang.llvm.org/extra/clang-tidy/checks/hicpp-use-equals-default.html

Defaulting/deleting constructors/destructors

```cpp
#include <fstream>

class FileResource {
    public:
        FileResource() = default;

        explicit FileResource(const std::string& name) {
            m_output.open(name, std::ios::out);
        }

        FileResource(const FileResource& other) = delete;
        FileResource(FileResource&& other) = default;

        ~FileResource() = default;

        auto operator=(const FileResource& other) -> FileResource& = delete;
        auto operator=(FileResource&& other) -> FileResource& = default;

    private:
        std::ofstream m_output;
};
```

Pass by const reference

```
#include "someType.h"

class Example {
    public:
        explicit Example(const Type& type) :
            m_type(type)
        { }

    private:
        Type m_type;
};
```

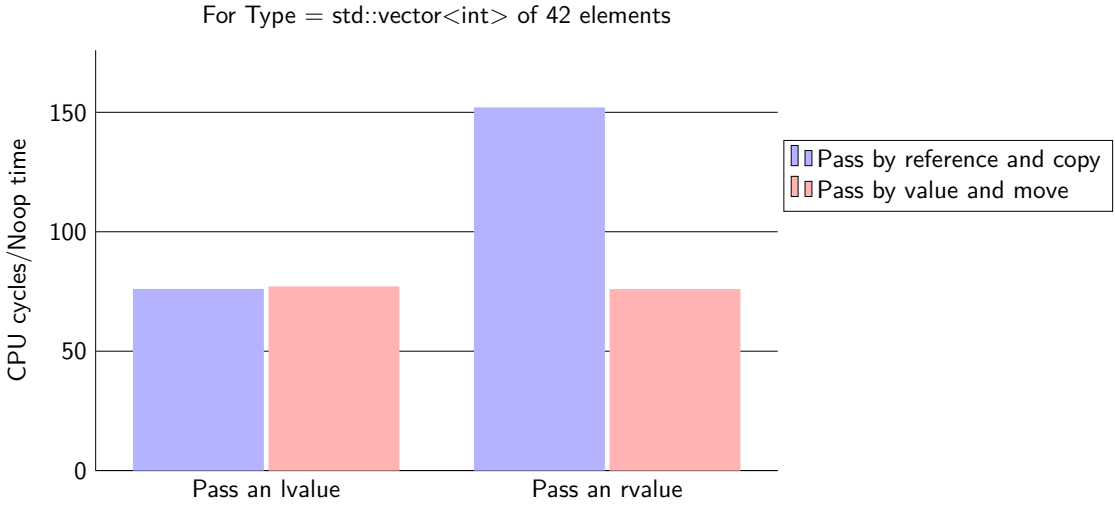
modernize-pass-by-value

Warning: pass by value and use std::move

Pass by value and move

```cpp
#include <utility>

#include "someType.h"

class Example {
    public:
        explicit Example(Type type) :
            m_type(std::move(type))
        { }

    private:
        Type m_type;
};
```

Benchmark



Fred Tingaud. *Quick C++ Benchmark*. URL: http://quick-bench.com/nj4kTFd-V85cTvn3ZOclFFW-r9o

Pass by value and move: summary

```
#include <utility>

#include "someType.h"

class Example {
    public:
        explicit Example(Type type) :
            m_type(std::move(type))
        { }

    private:
        Type m_type;
};
```

Allow copy elision

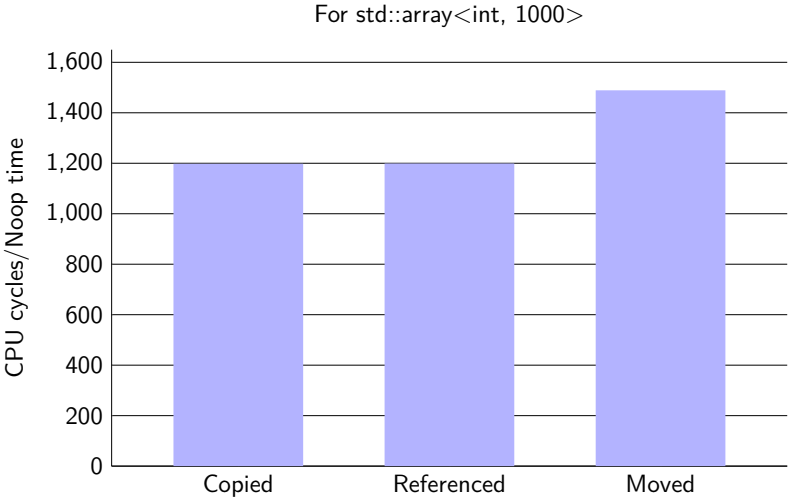
```cpp
constexpr size_t nbOfValues = 1000;
constexpr int value = 1;

auto fillArray() -> std::array<int, nbOfValues> {
    std::array<int, nbOfValues> result;
    std::fill(std::begin(result), std::end(result), value);
    return result;
}

auto fillArray(std::array<int, nbOfValues>& result) {
    std::fill(std::begin(result), std::end(result), value);
}

auto fillArrayMoved() -> std::array<int, nbOfValues> {
    std::array<int, nbOfValues> result;
    std::fill(std::begin(result), std::end(result), value);
    return std::move(result);
}

auto main() -> int {
    auto copied = fillArray();
    std::array<int, nbOfValues> referenced;
    fillArray(referenced);
    auto moved = fillArrayMoved();
}
```

clang-diagnostic-pessimizing-move

Warning: moving a local object in a return statement prevents copy elision

cppreference.com. *Copy elision*. URL: http://en.cppreference.com/mwiki/index.php?title=cpp/language/copy_elision&oldid=95959

Benchmark



Fred Tingaud. *Quick C++ Benchmark*. URL: http://quick-bench.com/0sXYjPQ0V9LA5ruDJnCAtXThnrI

Allow copy elision

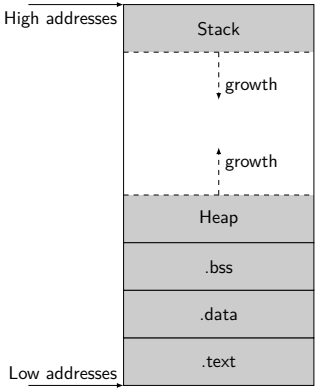
```cpp
#include <algorithm>
#include <array>
#include <utility>

constexpr size_t nbOfValues = 1000;
constexpr int value = 1;

auto fillArray() -> std::array<int, nbOfValues> {
    std::array<int, nbOfValues> result;
    std::fill(std::begin(result), std::end(result), value);
    return result;
}

auto fillArray(std::array<int, nbOfValues>& result) {
    std::fill(std::begin(result), std::end(result), value);
}

auto fillArrayMoved() -> std::array<int, nbOfValues> {
    std::array<int, nbOfValues> result;
    std::fill(std::begin(result), std::end(result), value);
    return std::move(result);
}

auto main() -> int {
    auto copied = fillArray();
    std::array<int, nbOfValues> referenced;
    fillArray(referenced);
    auto moved = fillArrayMoved();
}
```

Crash before main starts

```cpp
#include <exception>
#include <iostream>
#include <stdexcept>

auto getNumberOfDevices() -> unsigned int {
    throw std::runtime_error("Permission denied to devices in /dev");
}
class Devices {
    public:
        static auto nbOfDevices() -> unsigned int { return m_nbOfDevices; }
    private:
        static unsigned int m_nbOfDevices;
};
unsigned int Devices::m_nbOfDevices = getNumberOfDevices();

auto main() -> int {
    try {
        std::cout « "Number of devices: " « Devices::nbOfDevices() « std::endl;
    } catch(const std::exception& e) {
        std::cerr « "An error occurred" « std::endl;
    }
    return EXIT_SUCCESS;
}
```

```
$ ./a.out
terminate called after throwing an instance of 'std::runtime_error'
  what():  Permission denied to devices in /dev
abort (core dumped) ./a.out
```

Program memory layout



Crash before main starts

```cpp
#include <iostream>
#include <string>

namespace {
    const std::string EXAMPLE("example");
} // namespace


int main() {
    std::cout << EXAMPLE << std::endl;
    return EXIT_SUCCESS;
}
```

cert-err58-cpp

Warning: initialization of 'EXAMPLE' with static storage duration may throw an exception that cannot be caught

---

The Clang Team. *Clang-tidy - cert-err58-cpp*. URL: https://clang.llvm.org/extra/clang-tidy/checks/cert-err58-cpp.html

Software Engineering Institute - Carnegie Mellon. *ERR58-CPP. Handle all exceptions thrown before main() begins executing*. URL: https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main()+begins+executing

Crash before main starts

```cpp
#include <iostream>
#include <string>
#include <string_view>

#include <gsl/string_span>

namespace {
    using namespace std::literals;
    constexpr std::string_view solution1{"example"};     // since C++17
    constexpr auto solution2 = "example"sv;              // since C++17
    const gsl::czstring<> solution3 = "example";
} // namespace


int main() {
    std::cout << solution1 << std::endl;
    std::cout << solution2 << std::endl;
    std::cout << solution3 << std::endl;
    return EXIT_SUCCESS;
}
```

Problem statement

```cpp
#include <iostream>
class Base {
        int m_foobar;
    public:
        virtual ~Base() = default;
        auto getFoobar() -> int { return m_foobar; };
    protected:
        explicit Base(int foobar) : m_foobar{foobar} {}
};
class Child : public Base {
        int m_foo;
    public:
        explicit Child(int foo) : Base(foo), m_foo{foo} {}
        ~Child() override = default;
        auto getFoo() -> int { return m_foo;}
};
auto bar(Base& base) -> void { base = Child(2); }
auto main() -> int {
    Child child(42);
    bar(child);
    std::cout « "Foo = " « child.getFoo() « std::endl; // Expected: Foo = 2
    std::cout « "Foobar = " « child.getFoobar() « std::endl; // Expected: Foobar = 2
}
```

```
$ ./a.out
Foo = 42
Foobar = 2
```

Behind the scenes

```
class Base {
    public:
        virtual ~Base() = default;

        Base(const Base& other) = default;
        Base(Base&& other) = default;

        Base& operator=(const Base& other) = default;
        Base& operator=(Base&& other) = default;

        auto getFoobar() -> int { return m_foobar; };

    protected:
        explicit Base(int foobar) : m_foobar{foobar} {}

    private:
        int m_foobar;
};


// Since base is of type Base&, the compiler looks for:
//     Base& Base::operator=(const Child& other);   // Mis!
//     Base& Base::operator=(const Base& other);    // Hit!
base = Child(2);
```

Object slicing

```cpp
#include <iostream>
class Base {
        int m_foobar;
    public:
        virtual ~Base() = default;
        auto getFoobar() -> int { return m_foobar; };
    protected:
        explicit Base(int foobar) : m_foobar{foobar} {}
};
class Child : public Base {
        int m_foo;
    public:
        explicit Child(int foo) : Base(foo), m_foo{foo} {}
        ~Child() override = default;
        auto getFoo() -> int { return m_foo;}
};
auto bar(Base& base) -> void { base = Child(2); }
auto main() -> int {
    Child child(42);
    bar(child);
    std::cout « "Foo = " « child.getFoo() « std::endl; // Expected: Foo = 2
    std::cout « "Foobar = " « child.getFoobar() « std::endl; // Expected: Foobar = 2
}
```

cppcoreguidelines-slicing

warning: slicing object from type 'Child' to 'Base' discards 4 bytes of state

C++ Core Guidelines. *ES.63: Don't slice.* URL: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es63-dont-slice

Rule of three/five enforcement

```cpp
#include <string>

#include "someType.h"

class Three {
    public:
        ~Three() {
            std::remove(m_filename.c_str());
        };

    private:
        std::string m_filename;
};
```

hicpp-special-member-functions

Warning: class 'Three' defines a non-default destructor but does not define a copy constructor, a copy assignment operator, a move constructor or a move assignment operator

The Clang Team. *Clang-tidy - hicpp-special-member-functions.* URL: https://clang.llvm.org/extra/clang-tidy/checks/hicpp-special-member-functions.html

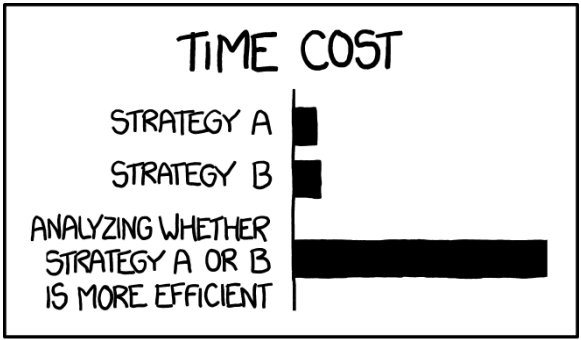
Rule of three/five enforcement

```cpp
#include <string>

#include "someType.h"

class Three {
    public:
        Three(const Three& other) = delete;
        Three(Three&& other) = default;

        ~Three() {
            std::remove(m_filename.c_str());
        };

        Three& operator=(const Three& other) = delete;
        Three& operator=(Three&& other) = default;

    private:
        std::string m_filename;
};
```

Object slicing protection

```cpp
#include <memory>
#include <utility>
class Base {
  public:
    virtual ~Base() = default;
  protected:
    Base() = default;
    Base(const Base& other) = default;
    Base(Base&& other) = default;
    Base& operator=(const Base& other) = default;
    Base& operator=(Base&& other) = default;
};

class Child : public Base {
  public:
    Child() = default;
    Child(const Child& /*other*/) = default;
    Child(Child&& /*other*/) noexcept = default;
    ~Child() override = default;
    Child& operator=(const Child& other) { type = other.type; return *this; }
    Child& operator=(Child&& other) noexcept { std::swap(type, other.type); return *this; }
  private:
    int type = 1;
};

void foo() {
  std::unique_ptr<Base> base1 = std::make_unique<Child>();
  std::unique_ptr<Base> base2 = std::make_unique<Child>();
  //*base2 = *base1;     // Compiler error!
}
```

Conclusion



xkcd. *Efficiency*. URL: https://xkcd.com/1445/

Thank you!

?