Core Guidelines

use modern C++ effectively

Stefan Kerkmann

Bjarne Stroustrup

> C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

Agenda

Errors and Maintainability Pitfalls
History of C++ Standards
Reasoning about a solution

—

Goals and Overview of C++ Core Guidelines
Applied Examples

—

Tools
Conclusion
Q&A

Errors

Memory Management
- Resource leaks
- Use-after free
- Double free

Concurrency
- Data races
- Deadlocks

Mutability

Uninitialized Variables

Maintainability

Weak Typing

Code Noise

Code Reuse

Referential Transparency

10^2 Language Features

10^3 Ways to solve a problem

to be continued...

C++ Standards

Year	C++ Standard	Informal name
1979	None	C with Classes
1983	*CFront 1.0*	C++ 1.0
1989	*CFront 2.0*	C++ 2.0
1998	ISO/IEC 14882:1998[23]	C++98
2003	ISO/IEC 14882:2003[24]	C++03
2011	ISO/IEC 14882:2011[25]	C++11, C++0x
2014	ISO/IEC 14882:2014[26]	C++14, C++1y
2017	ISO/IEC 14882:2017[9]	C++17, C++1z
2020	to be determined	C++20[17], C++2a

Looking for a solution to a preventable problem

Erik Naggum

> I believe C++ instills fear in programmers, fear that the interaction of some details causes unpredictable results [...] but the solution should have been to create and use a language that does not overload the whole goddamn human brain with irrelevant details.

Bjarne Stroustrup

> Within C++, there is a much smaller and cleaner language struggling to get out.

But no radical solution is viable

Sheer amount of code bases written since 1983

Keeping Backwards compatibility

Reshaping the whole language is not an option

C++ Core Guidelines - Goals

Use modern C++ features

Produce code that is

- Type safe 📎
- Exhibits no resource leaks ☔
- Catches common logic errors 👮
- Runs fast! 🏃
- Emphasizes simplicity and safety 👍

Suitable for gradual introduction into existing code bases

Be enforceable by machines 🤖

Overview

I: Interfaces

C: Classes and class hierarchies

R: Resource management

Per: Performance

E: Error handling

CPL: C-style programming

SL: The Standard Library

F: Functions

Enum: Enumerations

ES: Expressions and statements

CP: Concurrency and parallelism

Con: Constants and immutability

SF: Source files

T: Templates and generic programming

Showtime 

Con: Constants and immutability

Con.1: By default, make objects immutable

Con.2: By default, make member functions `const`

Con.3: By default, pass pointers and references to `const` s

Con.4: Use `const` to define objects with values that do not change after construction

Con.5: Use `constexpr` for values that can be computed at compile time

Bad 💩

```
struct
Person {
    DivineEternity
    LastJudgement(std::vector<Person*> family)
    {
        // Here be dragons
    }
};
```

Pass by value

Mutation of me or my family? 🧟

Might throw a `GoToHellException` 🔥

Better

```
struct
Person {
    DivineEternity
    LastJudgement(const std::vector<const Person*>& family)
    const noexcept
    {
        // Here be dragons
    }
};
```

Easier to reason about

A lot of code noise

Opt-in rather than Opt-out

R: Resource management

R.1 Manage resources automatically using resource handles and RAII

R.20 Use `unique_ptr` or `shared_ptr` to represent ownership

R.11 Avoid calling `new` and `delete` explicitly

R.1 Manage resources automatically using resource handles and RAII

Bad 💩

```
void func()
{
    File* d_fP = fopen("/etc/passwd", "r");
    if(d_fp == nullptr){
        // do error things
    }

    char line[512];
    while(fgets(line, sizeof(line), d_fp)) {
        // do things with line
    }

    {
        more code - maybe with early returns
        exceptions, threads ...
    }

    if(d_fp =/= nullptr){
        fclose(d_fp);
    }
}
```

Did we cover all cases and closed the file handle?

What about exceptions thrown in called procedures?

We will have a ressource leak 💧

Use RAII

```
struct SmartFP
{
    SmartFP(const char* fname, const char* mode)
    {
        d_fp = fopen(fname, mode);
    }
    ~SmartFP()
    {
        if(d_fp =/= nullptr) {
            fclose(d_fp);
        }
    }
    FILE* d_fp;
};
```

Aquire or allocate resource handle in constructor

Release resource handle in destructor

Lifetime of handle is bound to lifetime of the owning object ⏰

Better

```
void func() {
    SmartFP fp {"/etc/passwd", "r"};
    if(fp.d_fp == nullptr) {
        // do error things
    }

    char line[512];
    while(fgets(line, sizeof(line), fp.d_fp)) {
        // do things with line
    }

    {
        more code - maybe with early returns
        exceptions, threads ...
    }

    // note, no fclose
    }
```

👍 The FILE pointer will never leak due to destruction.

👍 Closing is deterministic and in a single place

💥 Vurnerable to "use after free" errors due to possible copy

R.20: Use `unique_ptr` or `shared_ptr` to represent ownership

Good `std::unique_ptr`

```
void func() {
    std::unique_ptr<FILE, int (*)(FILE*)> fp_uni(
        fopen("/etc/passwd", "r"),
        fclose
    );

    if(fp == nullptr) {
        // do error things
    }

    char line[512];
    while(fgets(line, sizeof(line), fp)) {
        // do things with line
    }

    {
        more code - maybe with early returns
        exceptions, threads ...
    }

    // note, no fclose
    }
```

- Single owner
- Copying is prohibited
- Can be moved
- Same size as raw pointer
- Automatically destroyed when out of scope
- Acts like a normal pointer

Good `std::shared_ptr`

```
void func() {
    std::shared_ptr<FILE> fp(
        fopen("/etc/passwd", "r"),
        fclose
    );

    if(fp == nullptr) {
        // do error things
    }

    char line[512];
    while(fgets(line, sizeof(line), fp)) {
        // do things with line
    }

    {
        more code - maybe with early returns
        exceptions, threads ...
    }

    // note, no fclose
    }
```

Multiple owners

Copying and moving is allowed

Keeps internal reference count to track usage

Size for counter, deleter and pointer

Acts like a normal pointer

R.11 Avoid calling `new` and `delete` explicitly

Okay

```
void func() {
    // setup

    auto taxi_1 = new Taxi(7.2e-2, 0.7, 75, 0.0);
    auto taxi_2 = new Taxi(12.5e-2, 0.95, 90, 0.0);

    std::vector<Taxi*> taxis;
    taxis.push_back(taxi_1);
    taxis.push_back(taxi_2);

    UI.start(taxis);

    // do bookings, fill up gas

    // Never forget to clean up...
    delete taxi_1;
    delete taxi_2;
}
```

Manual Memory Management is tedious and error prone

Unnessesary copies of pointers

Good

```
void func() {
    // setup

    std::vector<std::unique_ptr<Taxi>> taxis;
    taxis.emplace_back(
        std::make_unique<Taxi>(7.2e-2, 0.7, 75, 0.0)
    );
    taxis.emplace_back(
        std::make_unique<Taxi>(12.5e-2, 0.95, 90, 0.0)
    );

    UI.start(taxis);
    // do bookings, fill up gas

    // Everything is deleted automatically
}
```

Noisy but comfortable

Memory efficent

Clear ownership

No ressource leak 

Linters - Tooling

`clang-tidy` *CLion VS 2013+*

Clang-tidy has a set of rules that specifically enforce the C++ Core Guidelines. These rules are named in the pattern `cppcoreguidelines-*`.

`CppCoreCheck` *VS 2015+*

The Microsoft compiler's C++ code analysis contains a set of rules specifically aimed at enforcement of the C++ Core Guidelines.

Bonus!

`clang-format`

Automatic source code formatting following defined style rules.

Before

```
std::string Taxi::getState( ) const noexcept{
    std::ostringstream state;
    state << getName() << " >> "
    << std::fixed <<
    std::setprecision(2)
    << std::setfill(' ') <<
    std::setw(7) <<
    m_Mileage <<
    " km, " <<
    std::setw( 7 ) << m_GasLevel
    << " l, " << std::setw(
        7) << m_Balance
        << " Euro" << std::endl;
    return
    state.str();}
```

After

```
std::string Taxi::getState() const noexcept
{
    std::ostringstream state;
    state << getName() << " >> " << std::fixed << std::setprecision(2)
          << std::setfill(' ') << std::setw(7) << m_Mileage << " km, "
          << std::setw(7) << m_GasLevel << " l, " << std::setw(7) << m_Balance
          << " Euro" << std::endl;
    return state.str();
}
```

Conlusion

Writing safe, reliable and maintainable C++ is difficult

- Practice
- Care
- Experience

C++ has shortcomings that cannot be fixed soon

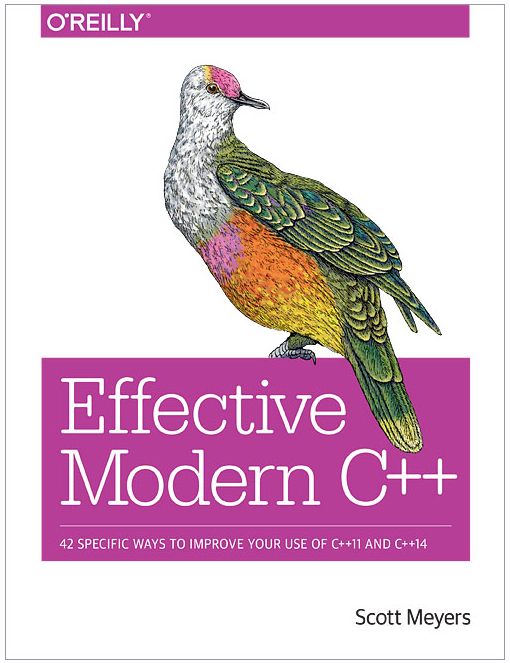
Guidelines and tools are there to help ☝️

Please use them

```
You won't regret it
```

Thank you for listening! 

Further Information



Scott Meyers: Effective Modern C++

C++ Core Guidelines

Modern C++ Features