Modernizing legacy C++ code

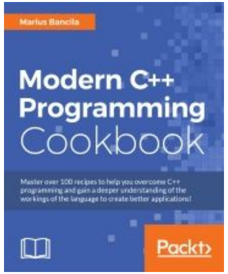
Marius Bancila

mariusbancila

marius.bancila

Microsoft Most Valuable Professional

MVP Reconnect



Agenda

- Short intro
- Legacy and modernization
- Good practices
  - Containers
  - Resource management correctness
  - Const correctness
  - Type casting correctness
  - Virtual correctness
- Q&A

What is legacy code?

Source:
https://www.reddit.com/r/ProgrammerHumor/comments/58lzgc/legacy_code/

“code inherited from someone else”

“code inherited from an older version of the software”

“code without tests”

“code that you wrote yesterday”

My experience with legacy code

- Projects started in mid-‘90 (Framework, ERP-CRM, tools)
- MFC, ATL, COM, .NET
- Very few unit and automated tests
- Files: 5000+ (4000+ C++), 4000+ (3500 C++)
- LOC: 2M (1.8M C++), 2M (1.9M C++)
- Classes: 6500 (5000 C++), 3000 (1800 C++)

What is modernization?

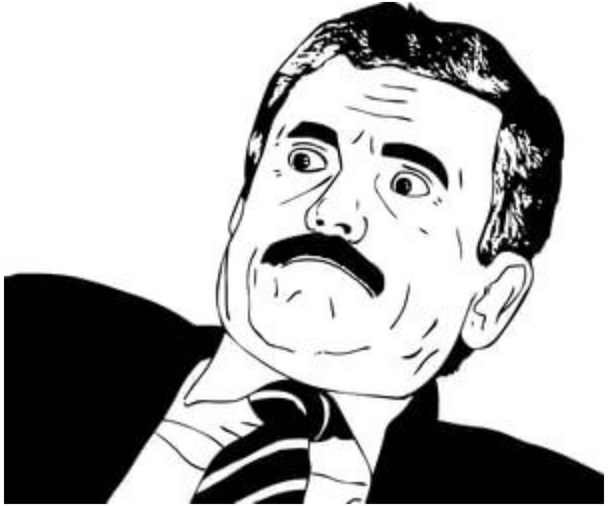
- Support for new software and hardware
- Architectural changes
- New tools
- New/modern frameworks and libraries
- New language and library features
- Principles, practices, and patterns
- Unit tests and automated tests
- Continuous integration

Containers

MFC vs Standard

- MFC containers
  - `CList`, `CArray`, `CMap`, ...
  - `CStringList`, `CDWordArray`, `CPtrArray`, ...
- Drawbacks
  - No performance guarantees
  - Don't work with standard algorithms
  - Don't work in range-based for loops *
  - Template unfriendly
  - Type unsafe
  - Backwards compatibility only

```cpp
CPtrArray arr;
arr.Add((void*)42);

Item item;
arr.Add(&item);

if(...) {
    CStringArray strarr;
    arr.Add((CPtrArray*)&strarr);
}
else {
    CDWordArray dwarr;
    arr.Add((CPtrArray*)&dwarr);
}
```



MFC vs Standard

- MFC containers
  - `CList`, `CArray`, `CMap`, ...
  - `CStringList`, `CDWordArray`, `CPtrArray`, ...
- Drawbacks
  - No performance guarantees
  - Don't work with standard algorithms
  - Don't work in range-based for loops *
  - Template unfriendly
  - Type unsafe
  - Backwards compatibility only

- Avoid using MFC containers
- Use standard containers by default
  - `std::vector` by default
- Advantages
  - Performance guarantees
  - Work with standard algorithms
  - Work in range-based for loops
  - Can be used in templates
  - Type safe

Debugging experience

```cpp
CPtrArray arr;
arr.Add(new Item{ 1, L"Item 1", 10.0 });
arr.Add(new Item{ 2, L"Item 2", 20.0 });
arr.Add(new Item{ 3, L"Item 3", 30.0 });

for (INT_PTR i = 0; i < arr.GetSize(); ++i)
   delete arr[i];
```

```cpp
std::vector<std::unique_ptr<Item>> arr;
arr.push_back(
   std::make_unique<Item>(1, L"Item 1", 10.0));
arr.push_back(
   std::make_unique<Item>(2, L"Item 2", 20.0));
arr.push_back(
   std::make_unique<Item>(3, L"Item 3", 30.0));
```

Name	Value
arr	<Information not available, no symbols loaded for mfc140ud.dll>
CObject	{...}
m_pData	0x00eb4910 {0x00eb4990}
m_nSize	3
m_nMaxSize	5
m_nGrowBy	0
arr.m_pData,3	0x00eb4910 {0x00eb4990, 0x00eb4550, 0x00eb4350}
[0]	0x00eb4990
[1]	0x00eb4550
[2]	0x00eb4350

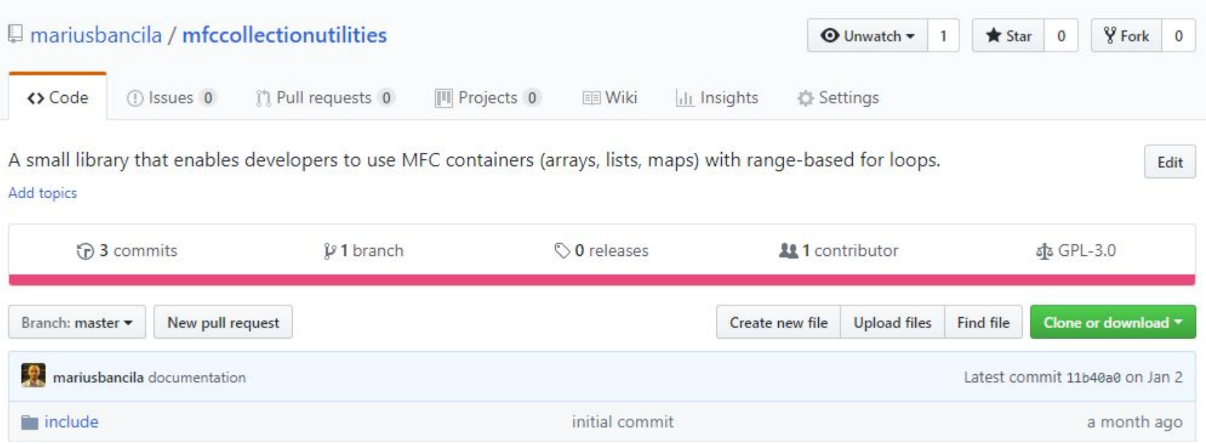
Name	Value
arr	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	unique_ptr {id=1 name=L"Item 1" value=10.000000000000000 }
[ptr]	0x008c27f8 {id=1 name=L"Item 1" value=10.000000000000000 }
id	1
name	L"Item 1"
value	10.000000000000000
[deleter]	default_delete
[Raw View]	{...}
[1]	unique_ptr {id=2 name=L"Item 2" value=20.000000000000000 }
[ptr]	0x008c2478 {id=2 name=L"Item 2" value=20.000000000000000 }
id	2
name	L"Item 2"
value	20.000000000000000
[deleter]	default_delete
[Raw View]	{...}
[2]	unique_ptr {id=3 name=L"Item 3" value=30.000000000000000 }
[ptr]	0x008c21f8 {id=3 name=L"Item 3" value=30.000000000000000 }
id	3
name	L"Item 3"
value	30.000000000000000
[deleter]	default_delete
[Raw View]	{...}

Range-based for loops for MFC containers

```
CArray<int> arr;
arr.Add(1);
arr.Add(2);
arr.Add(3);
arr.Add(4);
```

```
for (auto const n : arr)
{ /* do something */ }
```

https://github.com/mariusbancila/mfccollectionutilities



Resource management correctness

Using (smart & raw) pointers judiciously

Smart pointers vs raw pointers

Source:
http://pridenews.ca/2014/10/02/good-versus-evil-in-todays-world/

- Use `unique_ptr` and `shared_ptr` to model ownership
  - Use `make_shared()` and `make_unique()`
  - Use `weak_ptr` to break cycles
- Use raw pointers in non-owning semantics

- Pass smart pointers as function arguments only when you want to manipulate the smart pointer itself (share or transfer ownership)
- Pass objects by value, pointer (or const pointer), reference (or const reference)

Rules of Zero & Five

- Classes that declare custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership.
- Other classes should not declare custom destructors, copy/move constructors or copy/move assignment operators.

Rule of Zero - Martinho Fernandes / Scott Meyers

- Classes that define any of custom destructors, copy/move constructors or copy/move assignment operators should probably define them all

Rule of Five

RAII

The single easiest way to improve C++ code quality

James McNellis

Special member functions compiler rules

Explicitly declared	Default constructor	Copy constructor	Copy operator=	Move constructor	Move operator=	Destructor
nothing	YES	YES	YES	YES	YES	YES
Conversion constructor	NO	YES	YES	YES	YES	YES
Default constructor	NO	YES	YES	YES	YES	YES
Copy constructor	NO	NO	YES	NO	NO	YES
Copy operator=	YES	YES	NO	NO	NO	YES
Move constructor	NO	NO	NO	NO	NO	YES
Move operator=	YES	NO	NO	NO	NO	YES
Destructor	YES	Deprecated	Deprecated	NO	YES	NO

Passing unique_ptr as argument

value	`template <typename T>`
	`class foo {`
	`   std::unique_ptr<T> ptr;`
	`public:`
	`   foo(std::unique_ptr<T> p) : ptr(std::move(p)) {}`
	`};`
	
	`auto ptr = std::make_unique<int>(42);`
	`foo<int> f1(std::move(ptr));`
	`foo<int> f2(std::make_unique<int>(42));`
		• Transfers ownership
		• Two moves constructions
		• https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/

non-const l-value reference	`foo(std::unique_ptr<T> & p) :` `  ptr(std::move(p)) {}`	• May or may not transfer ownership
const l-value reference	`foo(std::unique_ptr<T> const & p) :` `{ /* use p */ }`	• Can use the pointer • Cannot transfer ownership
r-value reference	`foo(std::unique_ptr<T> && p) :` `  ptr(std::move(p)) {}`	• May or may not transfer ownership • One move construction • May not meet expectations • http://scottmeyers.blogspot.ro/2014/07/should-move-only-types-ever-be-passed.html

make_unique() / make_shared()

- `make_shared()`
  - C++11
  - Allocates the object and the control block in a single allocation
  - Avoids possible memory leaks in a particular scenario
- `make_unique()`
  - C++14
  - Consistency with make_shared()
  - Avoids possible memory leaks in a particular scenario

Memory leak scenario

```
int func_that_throws()
{
   throw std::runtime_error("oops...");
}

void do_something(std::unique_ptr<foo> p, int const v)
{
   /* use p and v */
}

// possible memory leak
do_something(std::unique_ptr<foo>(new foo), func_that_throws());
```

Memory leak scenario

```
int func_that_throws()
{
   throw std::runtime_error("oops...");
}

void do_something(std::unique_ptr<foo> p, int const v)
{
   /* use p and v */
}

// no memory leak
do_something(std::make_unique<foo>(), func_that_throws());
```

Memory leak scenario in C++17

```
int func_that_throws()
{
   throw std::runtime_error("oops...");
}

void do_something(std::unique_ptr<foo> p, int const v)
{
   /* use p and v */
}

// no memory leak in C++17
do_something(std::unique_ptr<foo>(new foo), func_that_throws());
```

§5.2.2 - Function call 5.2.2.4:

[...] Every value computation and side effect associated with the initialization of a parameter, and the initialization itself, is sequenced before every value computation and side effect associated with the initialization of any subsequent parameter.

Const-correctness

Making everything that should not change const

Const correctness

- `const` everywhere
  - Member functions
  - Function parameters
  - Objects
- `constexpr`
- Benefits for
  - Developers: better maintainability, better readability
  - Compiler: bugs detection, better optimizations in some cases
- Beware of
  - `auto` does not retain cv-qualifiers
  - `const_cast` removes cv-qualifiers
- Constant member functions and `mutable` specifier

const and mutable

```
struct point { double x; double y; };

class shape
{
   std::vector<point>            points;
   std::optional<double>         area;
public:
   void add_point(point const & p) {
      area.reset();
      points.push_back(p);
   }
   double get_area() const {
      if (!area.has_value()) {
         double a = 0;
         // expensive computation of the area
         area = a;          // ERROR
      }
      return area.value();
   }
};
```

const and mutable

```
struct point { double x; double y; };

class shape
{
   std::vector<point>            points;
   mutable std::optional<double> area;
public:
   void add_point(point const & p) {
      area.reset();
      points.push_back(p);
   }
   double get_area() const {
      if (!area.has_value()) {
         double a = 0;
         // expensive computation of the area
         area = a;           // OK
      }
      return area.value();
   }
};
```

const and mutable

```cpp
struct point { double x; double y; };

class shape
{
   std::vector<point>             points;
   mutable std::optional<double> area;
public:
   void add_point(point const & p) {
      area.reset();
      points.push_back(p);
   }
   double get_area() const {
      if (!area.has_value()) {
         double a = 0;
         // expensive computation of the area
         area = a;          // OK
      }
      return area.value();
   }
};
```

```cpp
class thread_safe_foo
{
   int                 data;
   mutable std::mutex  mt;

public:
   void update(int const d) {
      std::lock_guard<std::mutex> lock(mt);
      data = d;
   }

   int get() const {
      std::lock_guard<std::mutex> lock(mt);
      return data;
   }
};
```

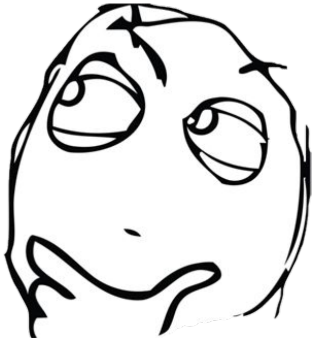
Type casting correctness

Using C++ casts

C-style casting

```
T* obj = (T*)expr;
```

What does this do?



```
1.  const_cast<T>(expr)
2.  static_cast<T>(expr)
3.  static_cast<T>(expr) + const_cast<T>(expr)
4.  reinterpret_cast<T>(expr)
5.  reinterpret_cast<T>(expr) + const_cast<T>(expr)
```

C++ casts

static_cast<T>(expr)	● Non-polymorphic types, including ○ Integrals to enums ○ Floating point to integrals ○ Pointer type to pointer type (no runtime checks)
dynamic_cast<T>(expr)	● Polymorphic types ○ Pointer or references between base and derived classes ○ Requires RTTI being enabled
const_cast<T>(expr)	● Types with different cv-qualifiers ● Only for objects not declared with cv-qualifiers (otherwise it’s UB) ● Does not translate to CPU instructions
reinterpret_cast<T>(expr)	● Bit reinterpretation, including ○ Integrals to pointer types and pointer types to integrals ○ Pointer type to pointer type (no runtime checks) ● Type unsafe ● Does not translate to CPU instructions

C++ casts

- Use C++ explicit casting instead of explicit type conversion (C-style casting)
- Benefits of C++ casts
  - better express user intent, both to the compiler and others that read the code
  - enable safer conversion between various types (except for `reinterpret_cast`)
  - can be easily searched for in source code

Source: http://www.heathceramics.com/

Virtual correctness

Always use virtual specifiers

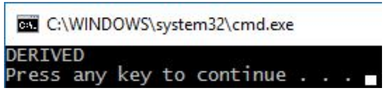
virtual, override, final

- `virtual` is optional in derived classes
  - But improves readability especially in deep hierarchies
- Always use `virtual`, `override`, and `final` to specify intent

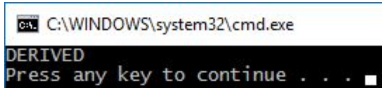
```
struct Base {
   virtual void foo() {}
};

struct Derived : Base {
   virtual void foo() override {}
};

struct Derived2 : Derived {
   virtual void foo() override final {}
};
```

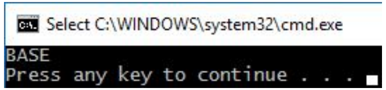
```
struct Derived3 final : Derived2 {
   virtual void foo() override final {}  // ERROR
};

struct Derived4 : Derived3 {               // ERROR
};
```

```cpp
struct MfcBase {
    virtual void DoSomething(DWORD arg)
        { std::cout << "BASE" << std::endl; }
};

struct MfcDerived : public MfcBase {
    virtual void DoSomething(DWORD arg)
        { std::cout << "DERIVED" << std::endl; }
};

void do_something(MfcBase* obj)
{
    obj->DoSomething(42);
}

MfcDerived obj;
do_something(&obj);
```

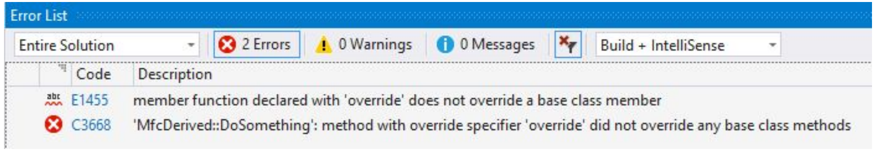


```cpp
struct MfcBase {
   virtual void DoSomething(DWORD_PTR arg)
      { std::cout << "BASE" << std::endl; }
};

struct MfcDerived : public MfcBase {
   virtual void DoSomething(DWORD arg)
      { std::cout << "DERIVED" << std::endl; }
};

void do_something(MfcBase* obj)
{
   obj->DoSomething(42);
}

MfcDerived obj;
do_something(&obj);
```






```cpp
struct MfcBase {
   virtual void DoSomething(DWORD_PTR arg)
      { std::cout << "BASE" << std::endl; }
};

struct MfcDerived : public MfcBase {
   virtual void DoSomething(DWORD arg) override
      { std::cout << "DERIVED" << std::endl; }
};

void do_something(MfcBase* obj)
{
   obj->DoSomething(42);
}

MfcDerived obj;
do_something(&obj);
```



Wrapping it up

- Use standard containers
- Use smart and raw pointers judiciously
- Use `const` on everything that should not change
    - `constexpr` on everything that could be evaluated at compile-time
- Use C++ casts
- Use `virtual`, `override`, and `final` specifiers

C++ CORE GUIDELINES

https://github.com/isocpp/CppCoreGuidelines

Q&A

Thank you!