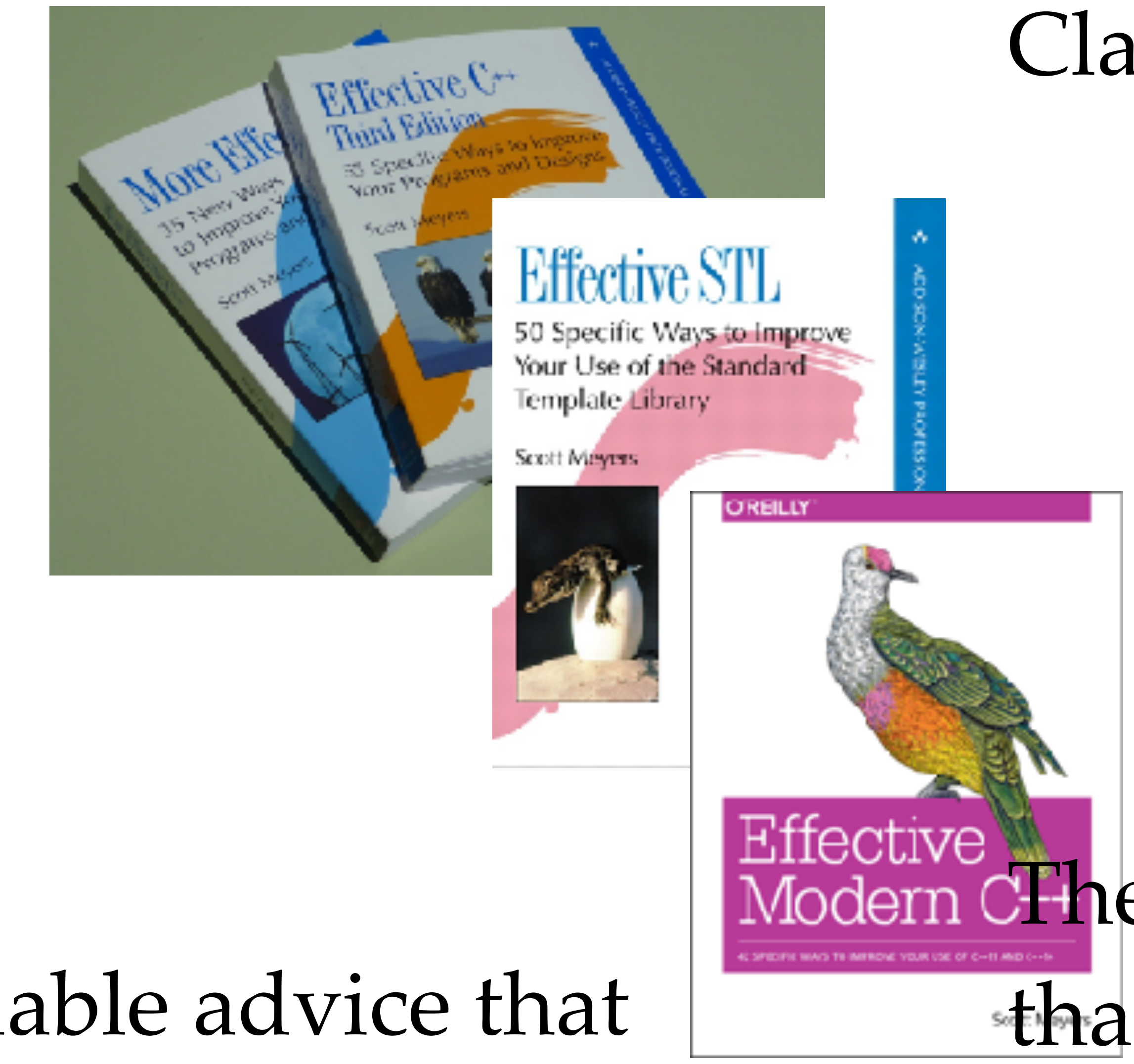
# Modernizing *Effective C++*

Presentation by Jon Kalb
Based the *Effective* C++ series
by **Scott Meyers**

# *Effective C++*

Scott Meyers' *Effective C++* series consists of four books:

*Effective C++*
*More Effective C++*
*Effective STL*
*Effective Modern C++*




All of these books contain valuable advice that we can assume any good software engineer has read and understood.

However the first three of these books were written for Classic C++.

These books contain valuable information about C++ that every Modern C++ software engineer should know and understand.

But they need to updated in small, but important ways.

That's what I *do*.

# Effective C++

Almost all of Scott's guidelines are still valuable (with some updating), but one piece of advice, great for Classic C++, is now just plain wrong for Modern C++.

Guideline:

~~Use the literal "0," not the macro "NULL" for nil pointers.~~

Modern Guideline:

Use nullptr, not the literal "0" nor the macro "NULL" for nil pointers.

# rvalue semantics

Will this compile?:

```
int a{0};
int b{23};
int c{42};

a + b = c;
```

**No**. The compiler will not allow us to modify temporaries (rvalues) of fundament types.

# rvalue semantics

Return-by-value is often useful:

  It allows for complicated expressions.

Arithmetic operators are a good example:

```
struct UPInt { ... };                                   // "unlimited precision integer”
                                                        // heap-based bits (vector<unsigned>)

UPInt operator+(UPInt const& lhs, UPInt const& rhs);
UPInt operator/(UPInt const& lhs, UPInt const & rhs);

UPInt a, b, c;
...
c = (a + b) / b;                                        // same as
                                                        // c.operator=(operator/(operator+(a,b), b))
```

# rvalue semantics

Will this compile?:

```
struct UPInt { ... };                                    // as before
UPInt operator+(UPInt const& lhs, UPInt const& rhs);     // as before
UPInt operator/(UPInt const& lhs, UPInt const& rhs);     // as before

UPInt a, b, c;
...
a + b = c;
```

Yes. It is the same as operator+(a, b).operator=(c).

Do we want it to compile?

How can we prevent it from compiling?

# return-by-const-value

Changes made to the value of a temporary, will be lost at the end of the expression, when the temporary is destroyed.

Modifying a temporary is *suspect* because the changes are going to be lost and the time/work to make them are wasted.

# Scott's *Effective* C++ advice: "*Do as the ints do.*"

Return-by-const-value can be useful:

- This is a way to make objects act like rvalues of fundamental types

```
struct UPInt { ... };                                                   // as before
UPInt const operator+(UPInt const& lhs, UPInt const& rhs);
UPInt const operator/(UPInt const& lhs, UPInt const& rhs);

UPInt a, b, c;
...
a = b + c;                          // normal usage works; const return value is implicitly converted to non-const
a + b = c;                          // same as operator+(a, b).operator=(c)
```

Because operator+ returns a `const` object, the last line won't compile!

Neither will this:

```
if (a + b = c) …                    // oops, used "=" instead of "=="
```

This could compile if an implicit UPInt ⇒ `bool` conversion exists.

# return-by-const-value

Changes made to the value of a temporary, will be lost at the end of the expression, when the temporary is destroyed.

Modifying a temporary is *suspect* because the changes are going to be lost and the time/work to make them are wasted.

Is there any time that we'd want to modify a temporary?

Howard Hinnant: *"You never want to modify a temporary,*
*except when you do."*

What data is held by the UPInt class?

We'd like to move the heap allocation of bits from a temporary `UPInt` rather than allocating again and copying the bits.

This is possible in Modern C++ with Move Semantics.

# guideline

- Return const objects when you want to emulate the rvalue semantics, of fundamental types,
  - but not if you want to enable move operations.

- If the type has resources that can be moved, we want to enable move operations,
  - so we don't want a const return type.

- Modern C++ provides us a way to create classes that support Move Semantics and also emulate the rvalue semantics of fundamental types.
  - We'll discuss topic again after covering the language features that make this possible.

# explicitly disallow use of implicitly generated member functions you don’t want

Consider a PascalArray class.

In Pascal, arrays are declared to have arbitrary upper and lower bounds.

Here is a skeleton:

```
template<class T>
struct PascalArray
{
   PascalArray( /* arguments */ );
   ~PascalArray();
   …
   private:
   …
};
```

# disallowing assignment

Assignment is not allowed for built-in arrays:

```
char string1[10];
char string 2[10];

string1 = string2;                    // error!
```

Assume you want to maintain this restriction:

```
PascalArray<int> intArray1( /* arguments */ );
PascalArray<int> intArray2( /* arguments */ );
intArray1 = intArray2;                // should be an error
```

- Usually if you don’t want to provide some functionality, you just don’t declare the function
- This won’t work for operator=, because C++ will generate the function automatically

# disallowing assignment - Classic C++

Classic C++ solution: declare the function private:

```
template<class T>
struct PascalArray
{
        ...
    private:
        PascalArray<T>& operator=(PascalArray<T> const& rhs);
        ...
};
```

This is good, but not perfect:

- Members and friends can still make assignments
- To prevent that, *don't define the function*. Uses of `operator=` will then generate link-time errors
- This trick was (in Classic C++) traditionally used in the iostream library to prevent users from accidently passing streams by value:
  - ➡ That requires disallowing use of the copy constructor.

# disallowing assignment - Modern C++

The Classic C++ approach to disallowing functions worked, but was a “hack” to work around a language limitation.

That limitation is addressed in Modern C++ with `delete`d functions.

`delete`d functions are the Modern C++ alternative to the Classic C++ approach for disallowing special functions.

This techniques is more powerful and more general than the Classic C++ approach.

# disallowing assignment - Modern C++

deleted functions are "defined," but can't be used.

- Most common application: prevent object copying:

```
struct Widget
{
   Widget(Widget const&) = delete;                  // declare and
   Widget& operator=(Widget const&) = delete;       // make uncallable
   …
};
```

- Note that Widget isn't movable, either.
  - Declaring copy operations suppresses implicit move operations!
  - It works both ways:

```
struct Gadget
{
   Gadget(Gadget&&) = delete;        // this also suppresses copy ops and move assignment
   …
};
```

# guideline

Explicitly disallow use of implicitly generated member functions you don't want.

# const member functions

Does C++ support self-modifying code?

Then aren't all C++ functions "const"?

What is const in a const member function?

```
struct MyType
{
        void MemberFunction(int a_parameter);                    // non-const member function
        void ConstMemberFunction(int a_parameter) const;         // const member function
};
```

What are the real parameters to these function?

```
struct MyType                                                        // pseudo code, not real C++
{
        void MemberFunction(MyType* this, int a_parameter);
        void ConstMemberFunction(MyType const* this, int a_parameter);
};
```

In a const member function, the implied "this" pointer is const.

"this" should really have been a reference instead of a pointer.

# ref-qualified member functions

Modern C++ takes the concept of qualifying "this" in member functions by support ref-qualified member functions.

```
struct MyType
{
    void MemberFunction(int a_parameter) &;      // #1 used if MemberFunction is called on an lvalue MyType
    void MemberFunction(int a_parameter) &&;     // #2 used if MemberFunction is called on a temporary (rvalue)
};

MyType MyTypeFactory();

MyType mt{MyTypeFactory()};
mt.MemberFunction(42);                           // Calls #1

MyTypeFactory().MemberFunction(23);              // Calls #2
```

If we assume that non-const member functions modify their object, this function is modifying a temporary.

# ref-qualified member functions

The expected use of ref-qualified member functions is to support polymorphic behavior such that we are copying from lvalue objects and moving from rvalue (temporary) objects.

```
struct MyType
{
    void MemberFunction(int a_parameter) &;          // Handle the copy-from-this case
    void MemberFunction(int a_parameter) &&;         // Handle the move-from-this case
};
```

This is not very often the case.

I call this pattern ***ref-implemented overloads*** because a unique implementation exists for both ref-qualified cases.

# rvalue semantics revisited

Remember UPInt?

- We wanted to emulate the rvalue semantics of fundamental types (prevent modification of temporaries)
- But we didn't want to inhibit Move Semantics

```
struct UPInt { ... };                                   // as before
UPInt const operator+(UPInt const& lhs, UPInt const& rhs);   // making const inhibits Move Semantics
UPInt const operator/(UPInt const& lhs, UPInt const& rhs);   // making const inhibits Move Semantics

UPInt a, b, c;
...
a + b = c;                  // won't compile; good!

if (a + b = c) …            // won't compile; good!

c = a + b;                  // can't use Move Semantics; bad!
```

# rvalue semantics revisited

Remember UPInt?

- We wanted to emulate the rvalue semantics of fundamental types (prevent modification of temporaries)
- But we didn't want to inhibit Move Semantics

- The problem here is that in Classic C++, we tried to solve the problem indirectly by having functions return constant values when the function doesn't really care if the value is modified or not.

- In Modern C++, we have tools to solve the real problem, modifying the temporary.
  - `delete`d functions
  - Ref-qualified non-static member functions

# rvalue semantics revisited

Remember UPInt?

- We wanted to emulate the rvalue semantics of fundamental types (prevent modification of temporaries)
- But we didn't want to inhibit Move Semantics

```
struct UPInt
{
    UPInt& operator=(UPInt const&) &;
    UPInt& operator=(UPInt const&) && = delete;     // This is not the ref-implemented pattern because
    UPInt& operator=(UPInt &&) &;                   // the rvalue case is not implemented.
    UPInt& operator=(UPInt &&) && = delete;
};
UPInt operator+(UPInt const& lhs, UPInt const& rhs);    // doesn't need to return const value so
UPInt operator/(UPInt const& lhs, UPInt const& rhs);    // Move Semantics are possible
UPInt a, b, c;
...
a + b = c;                  // won't compile; good!

if (a + b = c) …            // won't compile; good!

c = a + b;                  // can use Move Semantics; good!
```

# rvalue semantics revisited

Remember UPInt?

- We wanted to emulate the rvalue semantics of fundamental types (prevent modification of temporaries)
- But we didn't want to inhibit Move Semantics

```
struct UPInt
{
    UPInt& operator=(UPInt const&) &;
    // UPInt& operator=(UPInt const&) && = delete;        if the assignment operators are declared "&"
    UPInt& operator=(UPInt &&) &;
    // UPInt& operator=(UPInt &&) && = delete;            then the && versions need not be declared at all
};
UPInt operator+(UPInt const& lhs, UPInt const& rhs);     // doesn't need to return const value so
UPInt operator/(UPInt const& lhs, UPInt const& rhs);     // Move Semantics are possible
UPInt a, b, c;
...
a + b = c;                  // won't compile; good!

if (a + b = c) …            // won't compile; good!

c = a + b;                  // can use Move Semantics; good!
```

# guideline - modernized

- ~~Return const objects when you want to emulate the rvalue semantics, of fundamental types,~~
  - ~~but not if you want to enable move operations.~~

- Return non-const objects to support move semantics

- **R*ef-implemented* member functions should be in pairs of overloaded members declared *const &* and *&&*.**

- **Every non-static member function that is not ref-implemented should be qualified with either const or &.**
  - This prevents the modification of temporaries
  - It doesn’t inhibit Move Semantics

# modifying temporaries - revisited

Remember when we asked this question?

Is there any time that we'd want to modify a temporary?

Howard Hinnant: "*You never want to modify a temporary, except when you do.*"

The answer we gave was: *Move Semantics*!

There may be other cases where you want to modify a temporary. Consider:

```
takes_a_string(returns_as_string().trim());    // The member trim() modifies a temporary that
                                               // will be used before being destroyed.
```

# guideline - modernized - updated

- ~~Return const objects when you want to emulate the rvalue semantics, of fundamental types,~~
  - ~~but not if you want to enable move operations.~~

- Return non-const objects to support move semantics

- **R*ef-implemented* member functions should be in pairs of overloaded members declared *const &* and *&&*.**

- **Every non-static member function that is not ref-implemented should be qualified with either const or &.**
  - This prevents the modification of temporaries
  - It doesn't inhibit Move Semantics
  - *Unless you really want to support modification of temporaries, in which case it should be non-const, non-ref-qualified.*