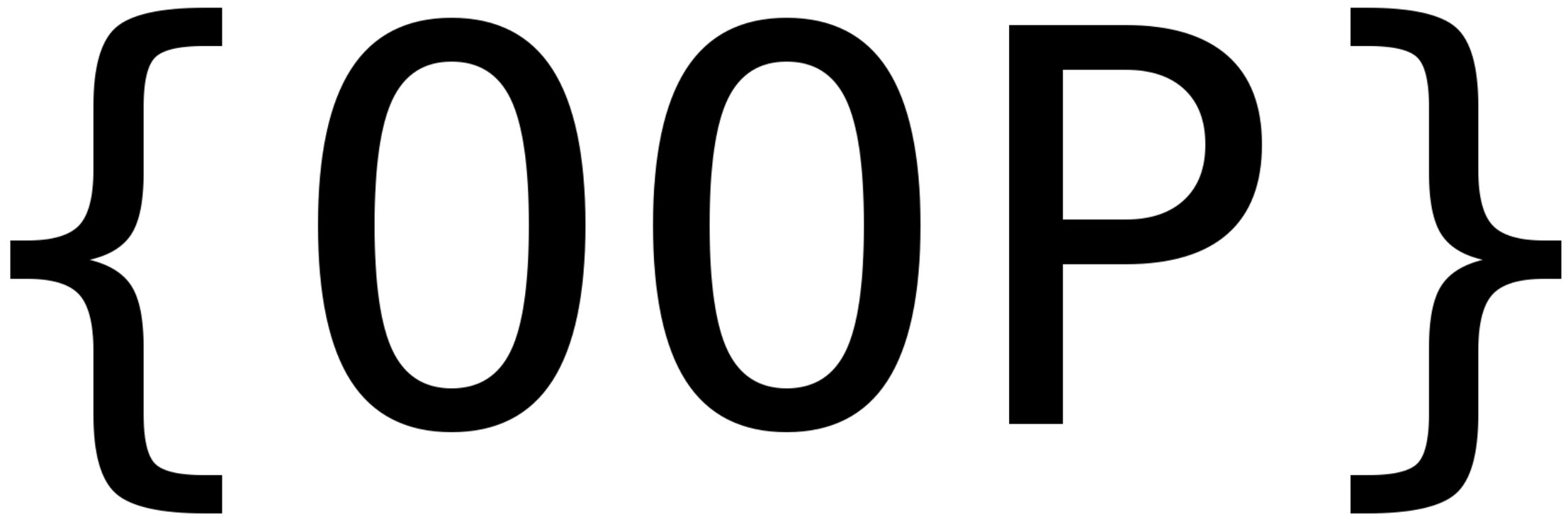


Back to Basics: Object-Oriented Programming

Presentation by Jon Kalb
CppCon 2019, 2019-09-17
thanks Andrei, Herb, & Scott

CULTURE / DEVELOPMENT

Why Are So Many Developers Hating on Object-Oriented Programming?

21 Aug 2019 12:00pm, by David Cassel

scope: what we will cover

- Objected-Oriented Programming best practices
  - theory of OOP
  - designing guidelines
    - inheritance and hierarchies
  - building (code-level) guidelines
    - creating and using virtual functions
    - runtime-time type information (RTTI) and dynamic_cast

scope: what we won't argue about

- definition of Object-Oriented Programming

a programming paradigm in C++ using polymorphism based on runtime function dispatch using virtual functions

OOP alternatives

Other paradigms might be better suited for any number of reasons.

object-based

static polymorphism

functional programming

taken as given that we'll use OOP

Now how do we do it best?

also won't argue about:
using vs typedef, East const,
intentional typing, ssize, et cetera

objects as libraries

- Derived objects in OOP can be thought of as independent libraries.
    - The libraries all have the same API.
        - This API is defined by the base class.
    - The libraries (objects) can implement and extend this API as they see fit.
        - compile time (the type of the derived object)
        - runtime (the state of the derived object)

simple example (logging)

- separation of concerns
  - application logic code that wants to log events or state
  - code that “knows about” and is responsible for logging
- base class: defines the API of logging libraries
- derived classes: provide different implementations of these libraries

simple example (logging)

```
struct Logger
{
 virtual void LogMessage(char const* message) = 0;
 virtual ~Logger() = default;
};

struct ConsoleLogger final: Logger
{
 virtual void LogMessage(char const* message) override
 {
  std::cout << message << '\n';
 }
};
```

base class: defines library interface

derived class: provides one library implementation

simple example (logging), continued

```
int main( )
{
 auto logger{std::make_unique<ConsoleLogger>( )};

 Logger* logger_ptr{logger.get( )};

 logger_ptr->LogMessage("Hello, World!");
}
```

without `virtual` function, would call `Logger::LogMessage`

static (compile-time) type of `logger_ptr` is `Logger*`

dynamic (runtime) type of `logger_ptr` is `ConsoleLogger*`

simple example (logging), continued

```
void LogHelloWorld(Logger& logger)
{
 logger.LogMessage("Hello, World!");
}

int main()
{
 auto logger{std::make_unique<ConsoleLogger>()};

 LogHelloWorld(*logger);
 LogHelloWorld(*static_cast<Logger*>(logger.get()));

}
```

code neither knows nor depends on dynamic type of logger

simple example (logging), continued

```
struct FileLogger final: Logger
{
 FileLogger(char const* filename):
  output_{filename} {}
 virtual void LogMessage(char const* message) override
 {
    output_ << message << '\n';
 }
private:
 std::ofstream output_;
};
```

simple example (logging), continued

```
void LogHelloWorld(Logger& logger)
{
 logger.LogMessage("Hello, World!");
}


int main()
{
    auto cl{ConsoleLogger{}};
    auto fl{FileLogger{"logfile.text"}};

    LogHelloWorld(cl);
    LogHelloWorld(fl);
}
```

from previous slide

Code written to the base type API can be used with any derived type.

Calling code can substitute objects of any derived type.

simple example (logging), continued

```
struct DialogBox final: Logger
{
  ~~~
};

struct Socket final: Logger
{
  ~~~
};
```

Liskov substitution

- Barbara Liskov defined what she called the Subtype Requirement.
- Essentially, D is a subtype of B if the provable properties of objects of type B are also provable properties of objects of type D.
- In practice, this is interpreted to mean that code written to behave correctly against the API defined by B will also work correctly for objects of type D if D is a proper subtype.
- Not all derived types are subtypes.
- consider:

Liskov substitution, an invalid subtype

```
struct SurpriseLogger final: Logger
{
 virtual void LogMessage(char const* message) override
 {
    std::exit(EXIT_FAILURE);
 }
};
```

fails to provide the semantics of the LogMessage API

OOP theory

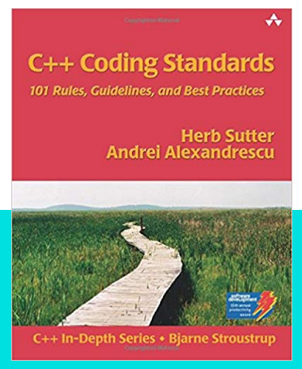
- Our model for OOP in C++ is that a hierarchy of derived types will represent a families of libraries that all implement the same API, which is defined by the base class.

- It is our assumption, as designers, and our goal, as implementers, to support Liskov Substitution which means that each derived type is a logical subtype of the base class, providing the intended semantics of the defined interface.

OOP theory

- From *C++ Coding Standards*
  - Rule 38: *Practice safe overriding.*

- "After the base class guarantees the preconditions and postconditions of an operation, any derived class must respect those guarantees. An override can ask for *less* and provide *more*, but it must never require more or promise less because that would break the contract that was promised to calling code."



implementation

Back to Basics: Virtual Dispatch and Its Alternatives

Inbal Levi

Thursday, 16:45

guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

Use the non-virtual interface (NVI) idiom.

inheritance

- Many early users of inheritance conceived of it as a technique for enabling code re-use.

- Inheritance should be applicable to any case of two code paths being similar, but not identical.

- This is likely to end in an unmaintainable mess.

inheritance

- The starting point or OOP as a problem solving tool:
- Public inheritance models “is-a”
  - Base class defines an interface for an object that might provide a type of functionality
  - Derived classes provide implementations of different expressions of that object type

simple example (logging)

```
struct Logger
{
 virtual void LogMessage(char const* message) = 0;
 virtual ~Logger() = default;
};

struct StatusDisplayer final: Logger
{
 virtual void LogMessage(char const* message) override
 {
  /* Show message as current status on LCD */;
 }
};
```

"Displayer" not strictly a logger.

inheritance

- Powerful hierarchies are built on well-defined abstractions

- Object types that do two or more only-partially related things
    - Make it hard to determine an optimal interface
    - Make it hard to write client code
    - Result in unmaintainable hierarchies

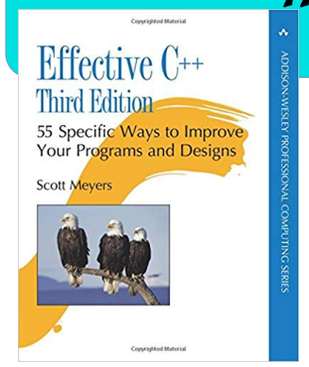
guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

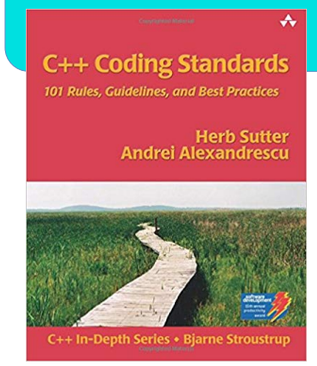
Effective C++: Item 32

“Make sure public inheritance models “is-a”.”



C++ Coding Standards: Item 37

“Public inheritance is substitutability. Inherit, not to reuse, but to be reused.”



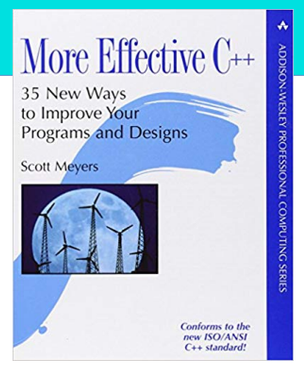
guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

Use the no NVI) idiom.

More Effective C++: Item 33 “Make non-leaf classes abstract”



Scott's challenge

```
Lizard liz1;
Lizard liz2;

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
~~~

*pAnimal1 = *pAnimal2;
```

slicing

```
sizeof(*pAnimal1)
```

not slicing, but logic error

the fundamental lie

- In OOP we are constantly lying to the compiler about object pointers
- The static type is almost never the dynamic type
- We survive by this rule:

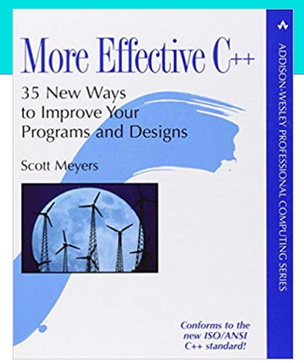
> Only dereference an OOP pointer/ref to access base class members.

- Because the (static) type, when dereferenced, is not the actual (dynamic) type.

Scott's solution

Make non-leaf classes abstract.

More Effective C++: Item 33 "Make non-leaf classes abstract"

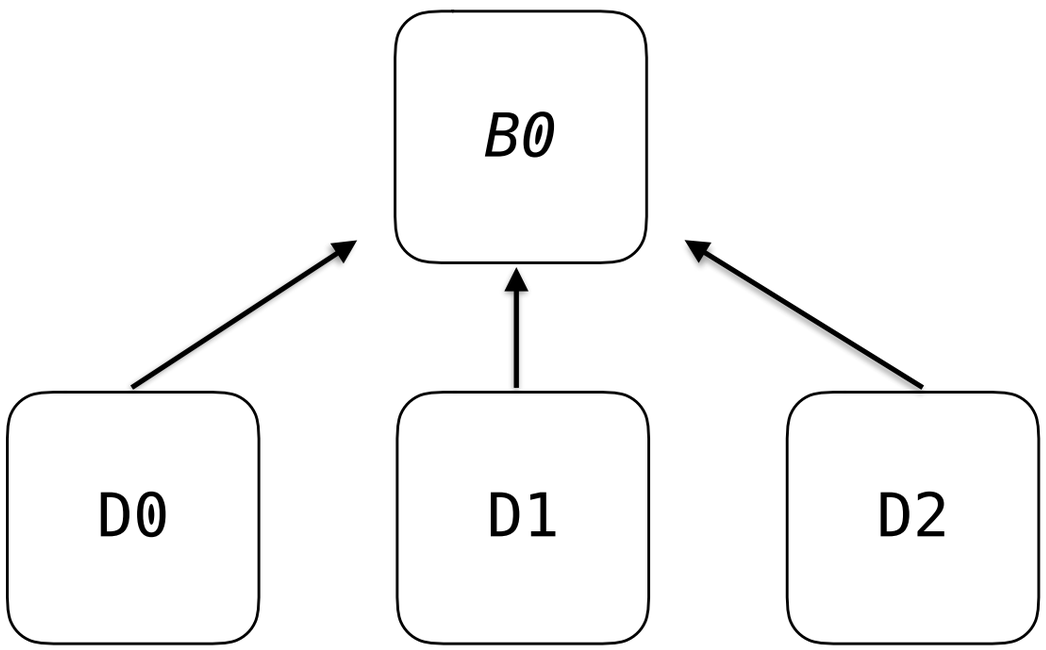


- Required to solve Scott's problem
- Results in better hierarchies

Scott's solution

- Step one:
  - Make every class in your hierarchy either a base-only or leaf-only

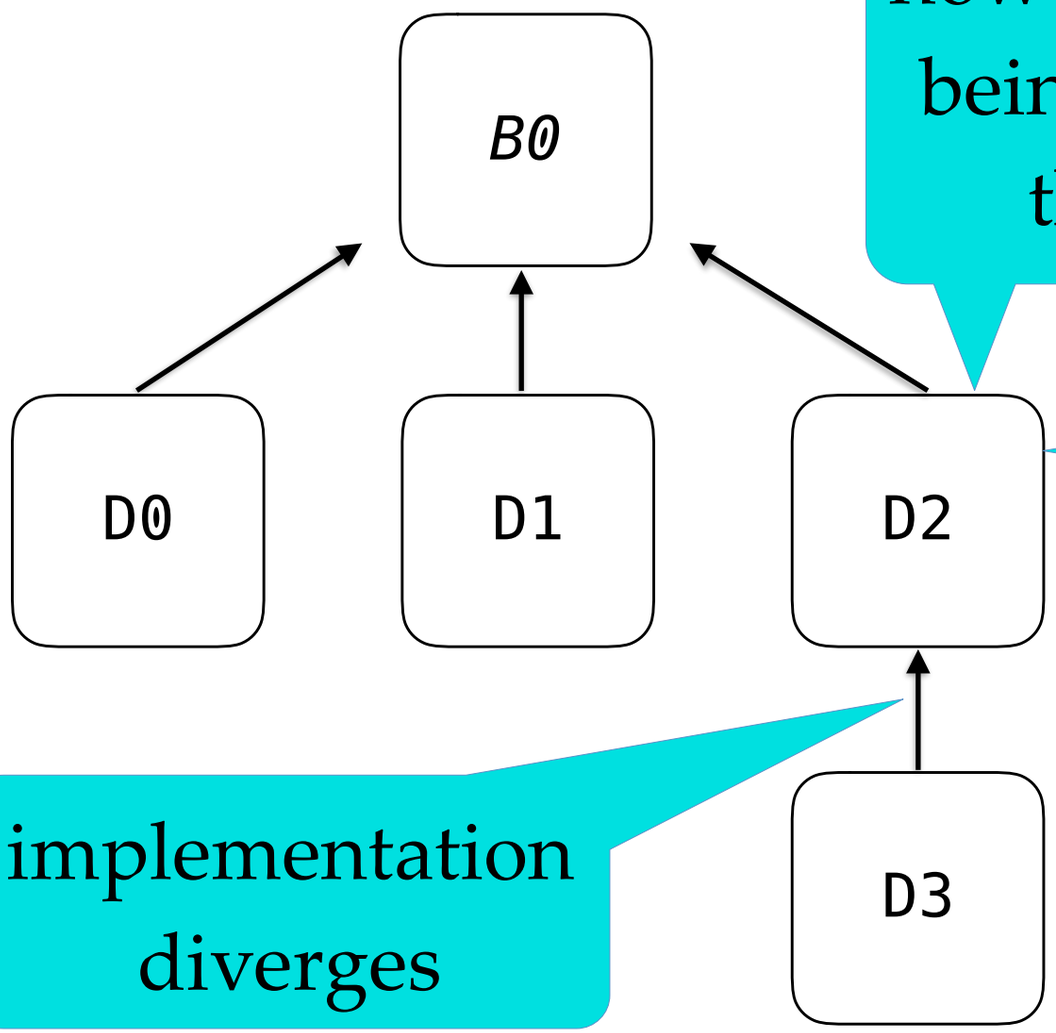
Scott's solution



inevitable

- We need a D3 which is exactly like D2, but with one little twist.

inevitable

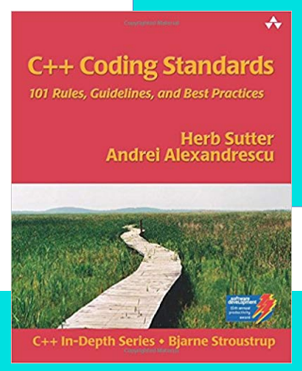


now has special logic for being D2 and for being the D3 base class

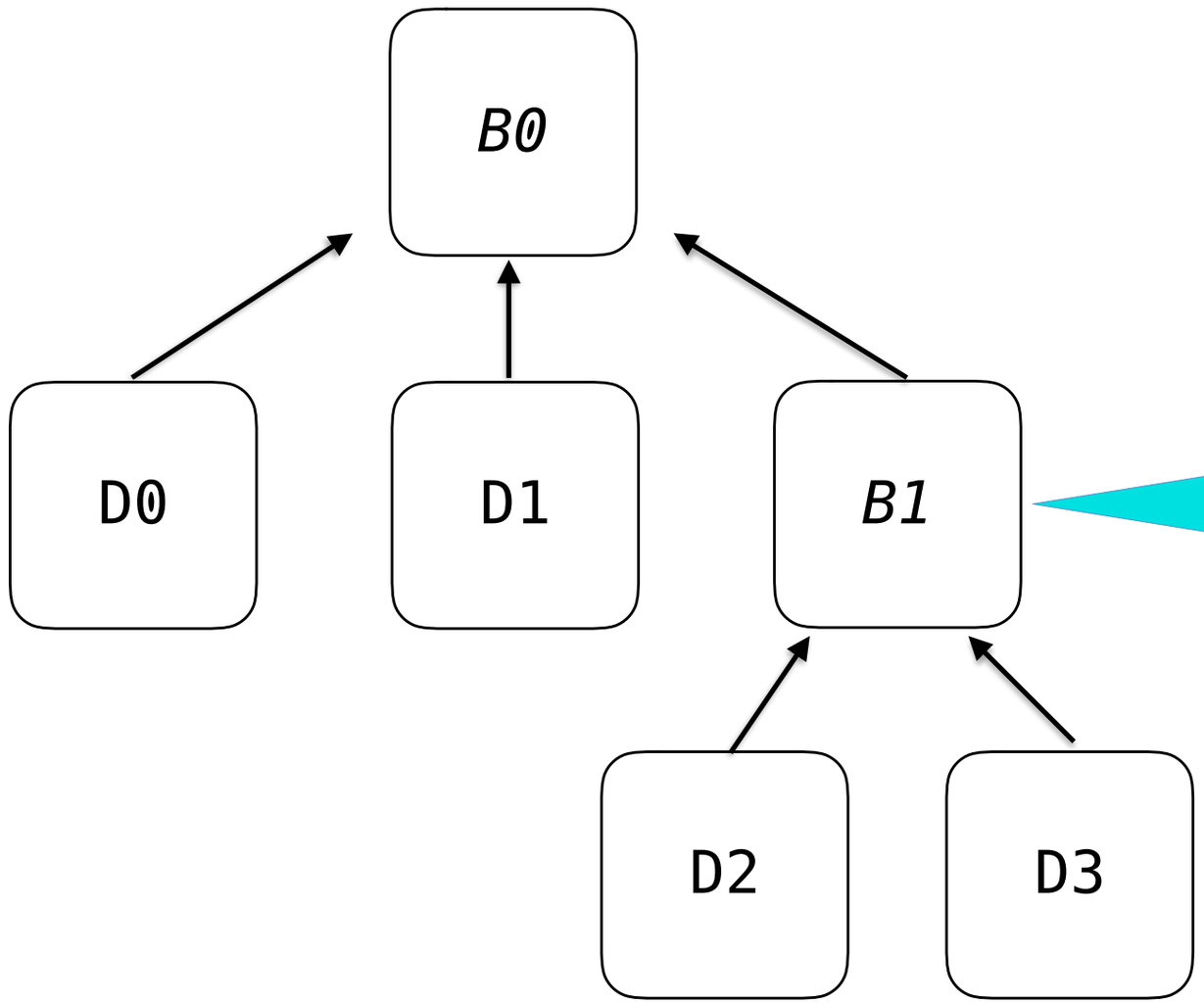
new requirements

implementation diverges

violates fundamental design guideline:
Item 5: Give one entity one cohesive responsibility.



Scott's solution



refactor:
new base class (B1)
features what is
common to both D2
and D3

robust in the face of
maintenance

Scott's solution

- Step one:
  - Make every class in your hierarchy either a base-only or leaf-only
- Step two:
  - Make bases
    - abstract (add one or more pure virtual functions)
    - protected assignment operators
- Step three:
  - Make leaf classes
    - concrete (override all pure virtual functions)
    - public assignment operators
    - *final*

solves the slicing problem

added by Jon

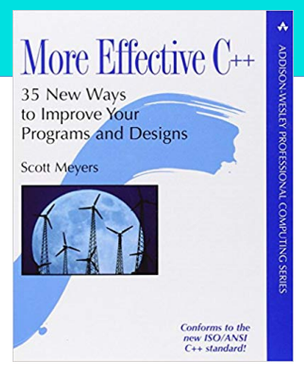
guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

Use the no NVI) idiom.

More Effective C++: Item 33 “Make non-leaf classes abstract”



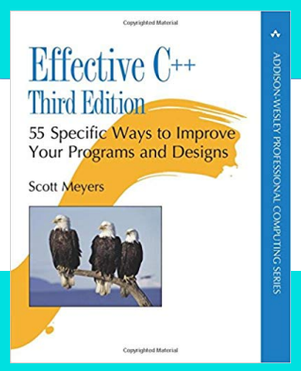
guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

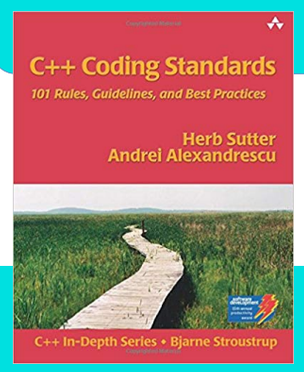
Make non-leaf classes abstract.

Use the non-virtual interface (NVI) idiom.

Effective C++: Item 35
“Consider alternatives to virtual functions.”



C++ Coding Standards: Item 39
“Consider making virtual functions nonpublic, and public functions nonvirtual.”



NVI idiom

```
struct Logger
{
 virtual void LogMessage(char const* message) = 0;
 virtual ~Logger() = default;
};

struct NVILogger
{
 void LogMessage(char const* message)
 { /* do prep */ DoLogMessage(message); /* do finish */ }
 virtual ~Logger() = default;
 private:
 virtual void DoLogMessage(char const* message) = 0;
};
```

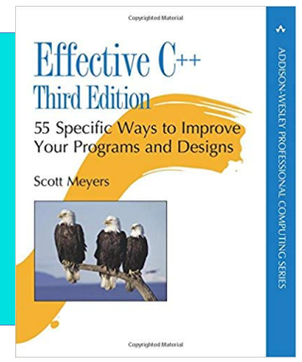
Public API is non-virtual. It calls non-public virtual to do work.

Much of the value in pre and post functionality. Can't be added later.

NVI idiom

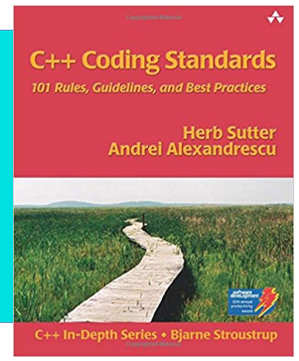
- Base class in control
  - enforce pre/post conditions
  - instrumentation, etc.
- Robust in the face of change
  - can add/remove pre/post processing without breaking callers or derivers

Scott got these.



- Each interface can take its natural shape.

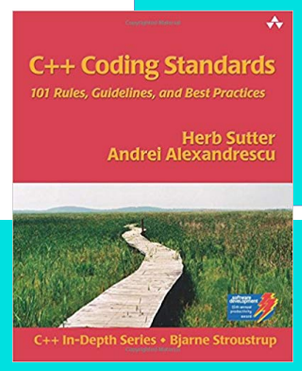
Most important!



NVI idiom

- Each interface can take its natural shape.
  - Public virtual functions have competing responsibilities for competing audiences:
    - specifies interface for callers
    - specifies implementation detail for derivers

violates fundamental design guideline:
Item 5: Give one entity one cohesive responsibility.



NVI idiom

- There are two audiences (callers and derives)
  - deserve two APIs
    - called (public interface)
    - derived (virtual interface)
  - no reason to expect that they will have similar
    - number of calls
    - parameters
    - return type

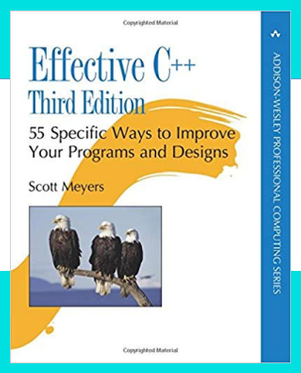
guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

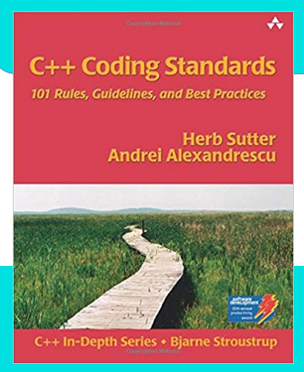
Make non-leaf classes abstract.

Use the non-virtual interface (NVI) idiom.

Effective C++: Item 35
“Consider alternatives to virtual funcions.”



C++ Coding Standards: Item 39
“Consider making virtual functions nonpublic, and public functions nonvirtual.”



guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

Use the non-virtual interface (NVI) idiom.

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

Use “override” for overridden functions.

Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtual functions in constructors or destructors.

Use dynamic rather than static casts for downcasting, but avoid casing by refactoring where possible.

non-virtual “overriding”

- Chris is using a legacy framework and doesn’t have write access to the framework code.
- They discover an error in a base class’s non-virtual function.
- Chris is desperate for a solution and writes an “override” function even though the base function isn’t virtual.
- Because Chris isn’t certain if this will work, they write a test function in the derived class.

scoping

```
struct base
{
    int erroneous() {return 0;}
};
struct derived final: base
{
    int erroneous() {return 1;}
    void test_erroneous() {assert(1 == erroneous());
                           std::cout << "success\n";}
};
int main()
{
    derived d;
    d.test_erroneous();
}
```

Will this test pass?

Is the fix successful?

scoping

```
struct base
{
    int erroneous(void) {return 0;}
};
struct derived final: base
{
    int erroneous(void) {return 1;}
    void test_erroneous() {assert(1 == erroneous());
                           std::cout << "success\n";}
};
int main()
{
    base* b{new derived};
    assert(1 == b->erroneous());
    delete b;
}
```

Will this test pass?

Is the fix successful?

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

Use “override” for overridden functions.

Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtual functions in constructors or destructors.

Use dynamic rather than static casts for downcasting, but avoid casing by refactoring where possible.

simple example (logging), continued

```
struct Logger
{
 virtual void LogMessage(char const* message) = 0;
};

struct FileLogger final: Logger
{
 FileLogger(char const* filename):
  output_{filename} {}
 virtual void LogMessage(char const* message) override
 {
    output_ << message << '\n';
 }
private:
 std::ofstream output_;
};
```

destructor not virtual
Don't do this.

derived class data member

simple example (logging), continued

```
int main( )
{
 Logger* logger_ptr{new FileLogger{"logfile.text"}};

 logger_ptr->LogMessage("Hello, World!”);

 delete logger_ptr;
}
```

Which destructor is called?

`FileLogger` data members cannot be cleaned up.

Undefined behavior even if `FileLogger` has no members.

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

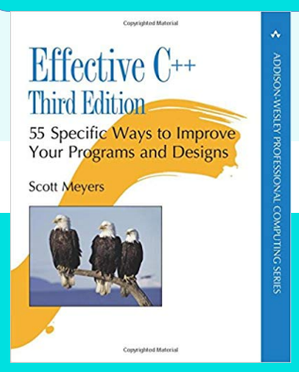
Use “override” for overridden functions.

Do not mix overloading and overriding.

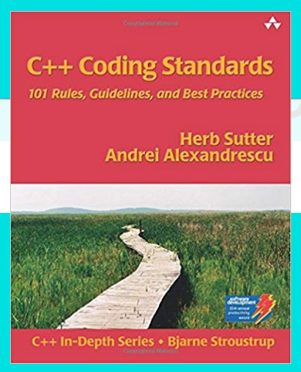
Don’t specify default values on function overrides.

avoid casing by ret

Effective C++: Item 7
“Declare destructors virtual in polymorphic classes.”



C++ Coding Standards: Item 50
“Make base class destructors public and virtual or protected and nonvirtual.”



simple example (logging)

```
struct Logger
{
 virtual void LogMessage(char const* message) = 0;
 virtual ~Logger() = default;
};

struct ConsoleLogger final: Logger
{
 virtual void LogMessage(char const* message) override
 {
  std::cout << message << '\n';
 }
};
```

`virtual` keyword optional, but recommended for overrides

simple example (logging), continued

```
struct FileLogger final: Logger
{
 FileLogger(char const* filename):
  output_{filename} {}
 void LogMessage(char * message)
 {
   output_ << message << '\n';
 }
private:
 std::ofstream output
};
```

No `virtual` keyword. Is `LogMessage( )` virtual?

No. `LogMessage( )` isn't an override.

Overrides must match signature exactly.

`const` is missing

simple example (logging), continued

```
struct FileLogger final: Logger
{
 FileLogger(char const* filename):
  output_{filename} {}
 void LogMessage(char * message) override
 {
    output_ << message << '\n';
 }
private:
  std::ofstream output_;
};
```

```
error: 'virtual void FileLogger::LogMessage(char*)' marked
'override', but does not override
```

override

- Using the "`override`" (contextual) keyword
  - Communicates to readers your intent
  - Is verified by the compiler

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

Use “override” for overridden functions.

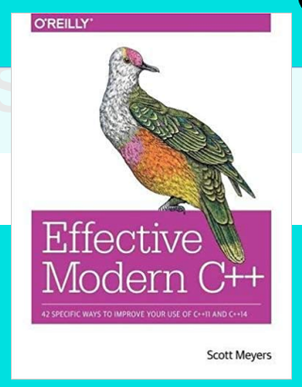
Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtu… …or destructors.

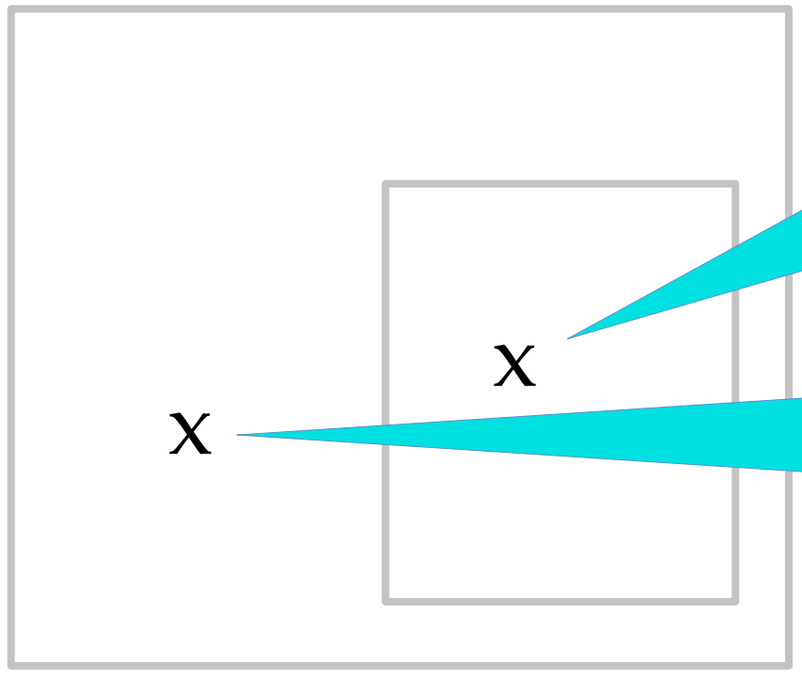
Use dynamic ra… …owncasting, but avoid cas… …refactoring where possible.

Effective **Modern** C++: Item 12
“Declare overriding functions `override`.”



scoping

- Inner scope declarations hide declarations in outer scope.



not an error to declare the same name in inner scope

outer scope declaration is hidden regardless of type of language element (object, type, function…)

scopes could be a type inside a namespace or another type

A derived type is “inner” to its base.

Any derived class declaration “hides” base declarations of the same name.

scoping

```
struct base
{
  using X = int;

};

struct derived final: base
{
  int X;

  std::vector<X> v;

};
```

`base::X` declared here as type alias

`base::X` in available here, but hidden by declaration of X as `int` data member.

compiler does not see the type X here because it is hidden by the data member declaration

scoping

- Function overloading rules:

1. look for called name in scope
2. if found
   - collect all candidates in scope
   - stop
3. else
   - move to next outer scope
   - go to #1

simplified:
actual rules complicated and include exceptions for ADL and "`using`"

- Overloading doesn't happen across scopes.

scoping

```
struct base
{
    auto foo(int x) {return 0;}
};

struct derived final: base
{
    auto foo(long x) {return 1;}
    auto foo(double x) {return 2;}
    derived()
    {
        std::cout << foo(0) << '\n';
        std::cout << foo(0L) << '\n';
        std::cout << foo(0.0) << '\n';
    }
};
```

What is the output of a call to `derived`'s default constructor?

Call is ambiguous. Overload set does not include `base::foo` because it is hidden by `derived::foo`.

scoping

```
struct base
{
    virtual int foo(int x) {return 0;}
};

struct derived final: base
{
    auto foo(long x) {return 1;}
    auto foo(double x) {return 2;}
    derived()
    {
        std::cout << foo(0) << '\n';
        std::cout << foo(0L) << '\n';
        std::cout << foo(0.0) << '\n';
    }
};
```

Making `base::foo` `virtual` changes nothing.

scoping

- This issue tends to come up when coders try to overload virtual functions.
    - This almost never works out as intended.
- It also comes up when a function declaration, intended to be an override, fails to exactly match the signature of the base virtual function.
    - This will be caught by using “`override`.”

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

Use “override” for overridden functions.

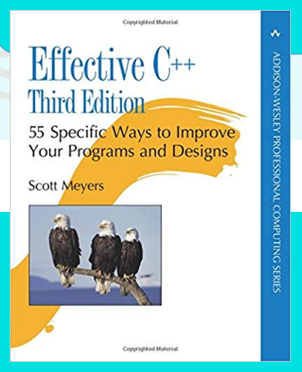
Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtua… or destructors.

Use dynamic ra… owncasting, but avoid cas… refactoring where possible.

Effective C++: Item 33
“Avoid hiding inherited names.”



default parameter values

- Chris is using a legacy framework and doesn't have write access to the framework code.
- They override a virtual function with a default parameter.
- Chris thinks the default value is ill-chosen, so they change it in the overriding declaration.
- Because Chris isn't certain if this will work, they write a test function in the derived class.

default parameter values

```
struct base
{
    virtual int bad_value(int i = 0) {return i;}
};
struct derived final: base
{
    virtual int bad_value(int i = 1) override {return i;}
    void test_bad_value() {assert(1 == bad_value());
                           std::cout << "success\n";}
};
int main()
{
    derived d;
    d.test_bad_value();
}
```

Will this test pass?

Is the fix successful?

default parameter values

```
struct base
{
    virtual int bad_value(int i = 0) {return i;}
};
struct derived final: base
{
    virtual int bad_value(int i = 1) override {return i;}
    void test_bad_value() {assert(1 == bad_value());
                           std::cout << "success\n";}
};
int main()
{
    base* b{new derived};
    assert(1 == b->bad_value());
    delete b;
}
```

Will this test pass?

Is the fix successful?

default parameter values

- Default parameter values are determined by the compiler by examining the declaration of the called function.
- They are inserted into the parameter list at the call site at compile-time.
- Because the actual function invoked is determined at runtime by address look up and not by the compiler, the passed default value will also be statically determined as the default parameter in the base class's function declaration.

default parameter values

- Specifying a default parameter in an overriding function will not cause a compile-time or runtime error, but it is likely to be confusing or misleading to readers of your derived class.

- If you specify the default parameter to the exact same value as the base class, this DRY violation has introduced a maintenance issue if the base class is ever modified.

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

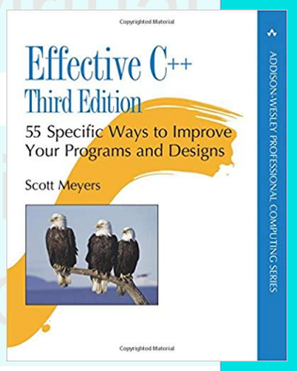
Use “override” for overridden functions.

Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtual[illegible]s or destructors.

Use dynam[illegible]downcasting, but avoi[illegible]e possible.



Effective C++: Item 37
“Never redefine a function’s inherited default parameter.”

simple example (logging)

```
struct Logger
{
 Logger( ) {LogMessage("Logger created");}
 virtual void LogMessage(c…nst* message) = 0;
 virtual ~Logger( ) = default;
};
```

constructor calls virtual function

```
~~~
 FileLogger fl{“logfile.text"};
```

What is the type of `fl` when in `Logger::Logger( )`?

construction and destruction types

- When a types's constructor or destructor is executing, the (static and dynamic) type of the object is that type.

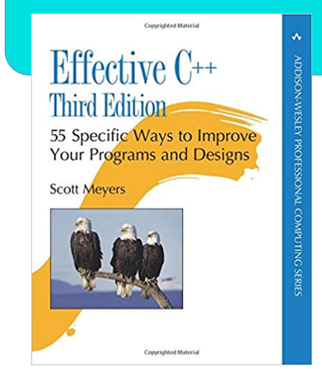
- This is true *even if the object being constructed (or destroyed) is a derived type*.

- Virtual function calls will ***not*** resolve to a function in a derived type during construction or destruction.

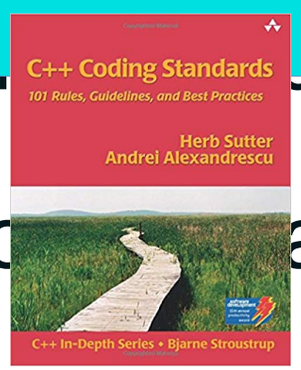
- This is a good thing. Derived type *data members are not yet constructed*.

guidelines / best practice: build

Effective C++: Item 9 “Never call virtual functions during construction or destruction.”



C++ Coding Standards: Item 49 “Avoid calling virtual functions in constructors and destructors.”



Don’t specify default values on function overrides.

Don’t call virtual functions in constructors or destructors.

Use dynamic rather than static casts for downcasting, but avoid casing by refactoring where possible.

simple example (logging)

- Different parts of our application are monitored by different parts of the IT department.

- Iff we are using the SMSLogger, we need to set the logger to notify a different “pager number” for different parts of our app.

simple example (logging)

```
struct SMSLogger final: Logger
{
 void SetCallee(PhoneNumber const&);
 virtual void LogMessage(char const* message) override;
};

~~~ in app code

 logger->SetCallee(duty_pager);
```

```
error: 'struct Logger' has no member named ‘SetCallee'
```

```
 static_cast<SMSLogger*>(logger)->SetCallee(duty_pager);
```

It compiles.
What's the problem?

upcasting

- Inheritance hierarchies are typically drawn with the base type at the top and derived types below the base (with an arrow pointed to the base of the "derived from" type).

- An ***upcast*** is casting a derived type pointer or reference to a pointer or reference to a type further ***up*** the hierarchy (to a base type).
    - Always safe, can be implicit, done *all the time*

- This works because an object of a derived type is always substitutable when a base type object is required.

downcasting

- A ***downcast*** is casting a base type pointer or reference to a pointer or reference to a type further ***down*** the hierarchy (to a derived type).

- A ***downcast*** is problematic because the compiler cannot know at compile time that the object ***is*** what we are casting to and a base type object cannot (in general) be substituted for a derived type object.

downcasting options

- One option is an *unconditional* (*static*) cast. If you are certain that the object in question is of the required type, this is *effective* and *efficient*, however:
  - The compiler cannot verify
  - Undefined behavior if you are incorrect
  - Uncertain in the face of code maintenance

- A better option is a *conditional*, *dynamic* cast. The compiler generates code to determine at runtime the object type.
  - Safe
  - Runtime overhead
  - Must code for the failure case

simple example with dynamic_cast

```
SMSLogger* sms_ptr{dynamic_cast<SMSLogger*>(logger)};
```

`dynamic_cast` may fail

`dynamic_cast` to pointer returns null on failure.

```
if (sms_ptr) sms_ptr->SetCallee(duty_pager);
```

requires null pointer check

```
~~~ or

    SMSLogger& sms{dynamic_cast<SMSLogger&>(*logger)};

    sms.SetCallee(duty_pager);
```

`dynamic_cast` to reference throws on failure, no checking required.

downcasting options: best

- The best option is to refactor our hierarchy in a manner that precludes any casting requirement.

- *If possible*, change the base class interface so that clients don't need to know the actual type. Client code allows derived classes to do the correct thing.

simple example (logging)

```
struct Logger
{
 virtual void SetDepartmentInfo(Dept const&) {}
 virtual void LogMessage(char const* message) = 0;
 virtual ~Logger() = default;
};
struct SMSLogger final: Logger
{
 virtual void SetDepartmentInfo(Dept const&) override
 {/* looks up on-call number and calls SetCallee() */}
 void SetCallee(PhoneNumber const&);
 virtual void LogMessage(char const* message) override;
};
~~~ in app code

 logger->SetDepartmentInfo(current_dept);
```

only overridden by `SMSLogger`

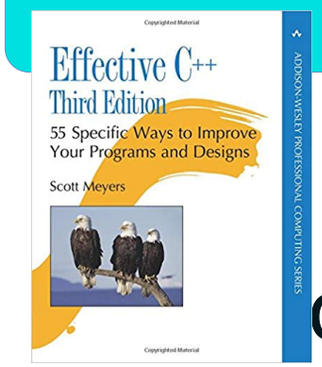
Only `SMSLogger` needs to implement a solution.

works for all types of `Loggers`

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Effective C++: Item 27
"Minimize casting."



se class destructors virtual.

for overridden functions.

Do not mix overloading and overriding.

on't specify default values on function overrides.

Don't call virtual functions in constructors or destructors.

Use dynamic rather than static casts for downcasting, but avoid casting by refactoring where possible.

guidelines / best practice: build

Overridden functions must be declared `virtual`.

Always make base class destructors virtual.

Use “override” for overridden functions.

Do not mix overloading and overriding.

Don’t specify default values on function overrides.

Don’t call virtual functions in constructors or destructors.

Use dynamic rather than static casts for downcasting, but avoid casing by refactoring where possible.

guidelines / best practice: design

Use OOP to model “is-a” relationships, not for code-reuse.

Make non-leaf classes abstract.

Use the non-virtual interface (NVI) idiom.