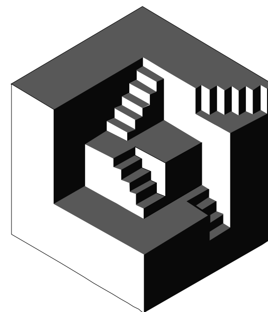
A Critical View of C++ Practices

Kevlin Henney
kevlin@curbralan.com

Agenda

- Intent
  - Reconsider some recommended C++ practices
- Content
  - C++ practices
  - Modularity
  - Class hierarchies
  - Object relationships
  - Control flow
  - In closing



C++ Practices

> *Do not use Hungarian notation.*
>
> **http://msdn2.microsoft.com/en-us/library/ms229045.aspx**

Idioms and Idiolects

- An idiom is a usage that is common within a community, i.e. part of the vernacular
  - An idiom is not by necessity and of itself good — some common practices are highly questionable
  - An idiom is not necessarily a pattern — many are simply conventions, not problem–solution pairings
  - A technique may be invented, but that does not make it an idiom — that is up to a community
- An idiolect is usage "common" to an individual
  - You want private data, not private language

The Evolution of Recommendations

- Recommendations change because...
  - They were ill founded, and time and reason have (finally) bettered them
  - Fashion, group think, marketecture and authority were responsible for the recommendations... and these are emergent and fickle, so they move on
  - Experience and reason reveal flaws in the recommendations or reveals better approaches
  - Underlying assumptions change, e.g. the presence of exceptions, threads or dynamically linked libraries

That Was Then...

```
class handle
{
public:
    handle &operator=(const handle &rhs)
    {
        if(this != &rhs)
        {
            delete body;
            body = new representation(*rhs.body);
        }
        return *this;
    }
    handle(const handle &);
    ~handle();
    ...
private:
    class representation
    {
        ...
    };
    representation *body;
};
```

An orthodox approach to dealing with the question of self-assignment is find in the absence of exceptions, but becomes lost when made exception safe by explicit addition.

```
class handle
{
public:
    handle &operator=(const handle &rhs)
    {
        if(this != &rhs)
        {
            representation *old_body = body;
            try
            {
                body = new representation(*rhs.body);
                delete old_body;
            }
            catch(...)
            {
                body = old_body;
                throw;
            }
        }
        return *this;
    }
    handle(const handle &);
    ~handle();
    ...
private:
    class representation
    {
        ...
    };
    representation *body;
};
```

But This Is Now

```
class handle
{
public:
    handle &operator=(const handle &rhs)
    {
        representation *old_body = body;
        body = new representation(*rhs.body);
        delete old_body;
        return *this;
    }
    handle(const handle &);
    ~handle();
    ...
private:
    class representation
    {
        ...
    };
    representation *body;
};
```

```
#include <algorithm>
class handle
{
public:
    handle &operator=(const handle &rhs)
    {
        handle copy(rhs);
        std::swap(body, copy.body);
        return *this;
    }
    handle(const handle &);
    ~handle();
    ...
private:
    class representation
    {
        ...
    };
    representation *body;
};
```

An approach based on stable intermediate forms — copying before release — unasks both the question of self-assignment and the question of exception safety. And in the absence of questions there is no need for decision-based control flow.

Unrecommending Recommendations

- Many recommendations that can be considered suspect are either unproven or lack rationale
  - Or both
- Many appear to be advocacy based on personal style preference or speculative generality
  - Words such as *flexible*, *reusable* and *general purpose* often used with insufficient qualification or restraint
  - Motivation for such recommendations often misses a root cause or a more effective alternative solution

Modularity

encapsulate *enclose (something) in or as if in a capsule.*

- *express the essential feature of (someone or something) succinctly.*
- *enclose (a message or signal) in a set of codes which allow use by or transfer through different computer systems or networks.*
- *provide an interface for (a piece of software or hardware) to allow or simplify access for the user.*

The New Oxford Dictionary of English

Modularity and Locality

- The principle of locality is a significant consideration in design
  - It aligns well with human cognitive abilities
  - Any reduction in locality should not be taken lightly
- Cohesion represents the alignment between the locality and the separation of modular parts
  - Modules are conventionally bound packaging units
- Uncohesive parts undermine locality
  - E.g. the role(s) of *realloc*, *<stdlib.h>*, *<iterator>*, *std::tr1*

Modularity in C++

- C++ has no formal module system
  - Informally, the source file and a corresponding header file fulfil this role
  - Similarly, dynamically linked libraries play the role of deployment modules, i.e. components
  - In other words, classic modules are a matter of convention and environment in C++
- C++ does support a variety of formal elements that offer a range of modularity to amodularity
  - Namespaces, classes and functions

The Naming of Space

- In many ways, namespaces can be considered wannabe modules
- But there is a lack of consensus concerning what should be covered by a namespace
  - A whole library, subsystems, separate modules, ...?
- The standard sets a poor example...
  - The packaging of the whole in namespace *std* is fine
  - But then there's *std::rel_ops* and the failure to deal with versioning, i.e. *std::tr1* and *std* in C++0X

Unstable Practices

- Beware of practices adopted from outside C++
  - E.g. Java's Internet domain convention for packages
  - E.g. .NET's namespacing based on deep nesting and long names
- Beware of partitioning namespaces based on uncohesive criteria
  - E.g. putting all constants or *typedef*s together
- Beware of using names that are unstable or reduce readability when used in action

Stable Practices

- Things to avoid...
  - Namespaces acting as modules to wrap lone classes
  - Namespaces for integrated subsystems and layers
  - Names incorporating company and product names
- Things to adopt...
  - Namespaces that wrap function-based headers — should be considered closed, not open
  - Multi-file namespacing should be reasonably flat
  - Names and partitions based on stable concepts

Non-Member Functions

- Some functions are best expressed as global
  - Certain operators that rely on conversions
  - Named function templates that are orthogonal with respect to type rather than bound to a specific type
- However, introducing globals much beyond this leads to a loss of (not rise in) encapsulation
  - Loss of cognitive chunking, readability, traceability, enforcement of constraints, etc.
  - Encapsulation is not just about data hiding

The ADL Uncertainty Principle

- In C++, ADL (Argument Dependent Lookup)...
  - Obscures a program's modular structure through loss of containment and loss of simple traceability
  - Is embarrassingly complicated, ridiculously subtle and significantly more of a problem than a solution
- The Interface Principle is not really a principle
  - Just mentioning a type in a function declaration does not make automatically make the function a part of the type's interface, even if supplied together — this is not how interfaces and modules are understood

Unnecessary Options

- Sometimes an interface tries to be too helpful but actually offers nothing extra
  - Variations that have no increase in expressiveness or capability should be excluded rather than included

```
class year {...};
enum month {...};
class date
{
public:
    date(year, month, int day_in_month) ...
    date(int day_in_month, month, year) ...
    date(month, int day_in_month, year) ...
    ...
};
```

Testing, Testing, One, Two, ...

- Separate unit tests from the code under test
  - Code containing its own tests may seem like a good idea, but introduces coupling and weakens cohesion
  - Tests should be strictly layered with respect to what they test — so, no *#ifdef TESTING* in target code
- But testing should not be separate from coding
  - Locality and immediacy of programmer testing and coding is valuable, i.e. tight, low-cost feedback loop
  - The spacing needed is between perspectives and levels, e.g. system testing separated from unit testing

Class Hierarchies

The history of all hitherto existing society is the history of class struggles.

Karl Marx and Friedrich Engels

Inheritance Tax

- Inheritance is the strongest form of coupling in a C++ object system
  - *friend* introduces a restricted form of coupling — one that can be pernicious, but it is not actually as strong
- Inheritance has a significant cognitive overhead
  - And in spite of much advice to reduce coupling, cut dependencies and use delegation, many current recommendations do the opposite
  - Class hierarchies can become so jammed with purpose that they lead to (develop)mental gridlock

Separation of Concerns

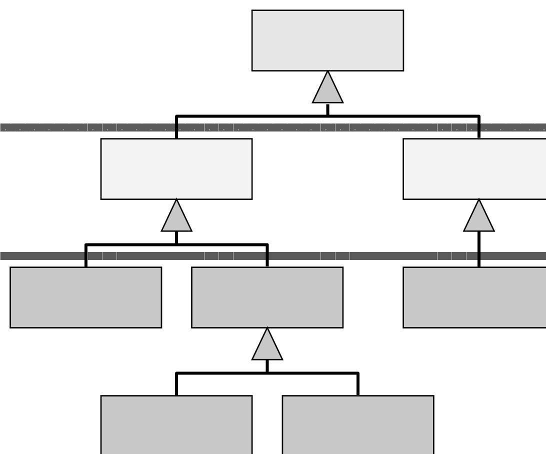
- Orthogonality is often the key to unblocking the obstacles arising from information overload
  - Therefore, favour Strategy and other pluggable approaches that push behaviour out of a class hierarchy over those that push behaviour down, such as Template Method
- Split classes and hierarchies along role lines
  - E.g. eliminate cyclic dependencies in class hierarchies by separating the role of creation from the root of a hierarchy into a separate factory

Infrastructure + Services + Domain

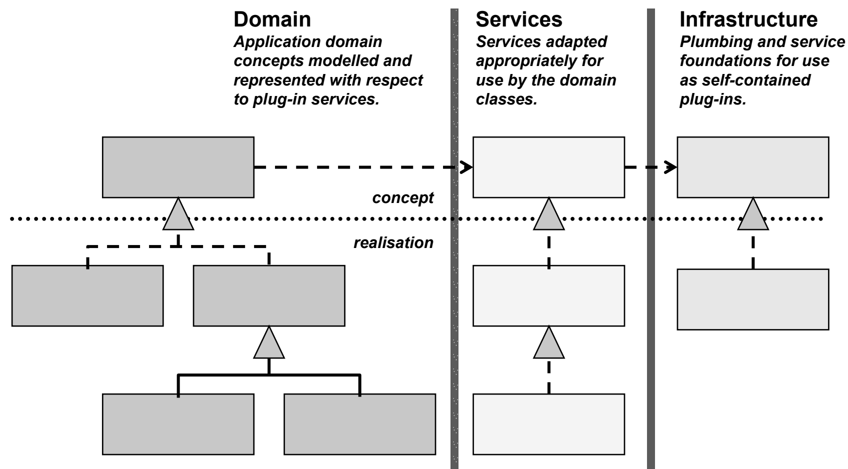
Infrastructure
Plumbing and service foundations introduced in root layer of the hierarchy.

Services
Services adapted and extended appropriately for use by the domain classes.

Domain
Application domain concepts modelled and represented with respect to extension of root infrastructure.



Domain × Services × Infrastructure



virtual Reality

- It has been suggested that *virtual*s should be *private* if they are only for use by the base
  - Note that this practice has no compiler enforcement
- However, this style should be avoided
  - It interferes with the more valuable and common advice that the *public* section of the class is for general users, the *protected* section is for derived dependents and the *private* section is... well, MYOB
  - The language rules that permit it are a constant and unnecessary surprise to non-guru programmers

The Ideal Class Hierarchy

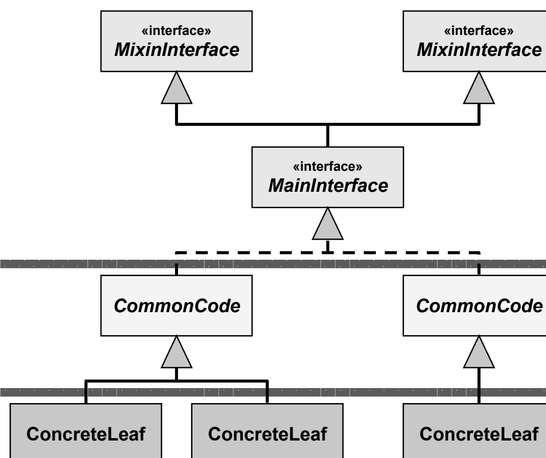
- The root of a class hierarchy should ideally be wholly abstract, i.e. an interface class
  - And this means no "helpful" default implementation — C++ developers find the temptation hard to resist
- Furthermore, *virtual* function definitions should ideally not be overridden
  - Introduce a *virtual* function as pure *virtual* and provide it with one definition in any branch of a hierarchy — avoid uninheriting behaviour
  - Builds on advice to inherit only from abstract classes

Pure Roots, Concrete Leaves

Pure Interface Layer
Interfaces may extend interfaces, but there is no implementation defined in this layer.

Common Code Layer
Only abstract classes are defined in this layer, possibly with inheritance, factoring out any common implementation.

Concrete Class Layer
Only concrete classes are defined, and they do not inherit from one another.



Non-*virtual* Interfaces?

- NVI suggests *virtual*s should be *private* and wrapped in *public* non-*virtual*s
  - Over the last decade or so this has been proposed by some as a good practice guideline
  - It has structural similarities with Template Method, but has a distinct form and (in)distinct motivation
- However, it is a solution in search of a problem
  - On close inspection the problems it purports to resolve are better addressed by other more mature and proven techniques, e.g. the Interceptor pattern

Non-Valuable Idiom

- NVI lacks either a clear motivation or a clear description of benefits
  - Motivation is often presented in terms of shotgun speculation — code simplification, instrumentation, extensibility, decoupling, thread safety, assertion checking, etc. — that does not stand up to scrutiny
- NVI singularly fails to adequately offer the benefits that are advertised as its motivation
  - In practice it is a somewhat tedious and verbose technique that adds baggage to a class hierarchy

Less Is More

```
class command
{
public:
    void execute()
    {
        do_execute();
    }
    void rollback()
    {
        do_rollback();
    }
    virtual ~command();
private:
    virtual void do_execute() = 0;
    virtual void do_rollback() = 0;
};
```

```
class command_processor
{
public:
    void execute(command *);
    void rollback();
    ...
};
```

```
class command
{
public:
    virtual void execute() = 0;
    virtual void rollback() = 0;
    virtual ~command();
};
```

```
class command_processor
{
public:
    void execute(command *);
    void rollback();
    ...
};
```

In a class hierarchy there are multiple possibilities for extension, including inheritance, delegation and management, e.g. adding Interceptors generally on the Command Processor or Decorators per instance on the Command. Adding behaviour via ad hoc modifications to the root class, sometimes via **#ifdef*****, clutters rather than clarifies.***

Gozer the Discreet

- Common wisdom and advice requires that base classes have *public virtual* destructors
  - This recommendation is well meant but incomplete
- However, making a destructor *public* suggests that deletion via a base is a necessary feature
  - For many base classes a *protected* — and not necessarily *virtual* — destructor results in both simplification and safety
  - E.g. interface classes that express a domain model and a manager that realises the model privately

Object Relationships

> *Increasingly, people seem to misinterpret complexity as sophistication, which is baffling — the incomprehensible should cause suspicion rather than admiration. Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user.*
>
> **Niklaus Wirth**

Essential Dependencies

- Encapsulation focuses on the use and quality of interfaces, abstracting unnecessary detail
  - "Program to an interface not to an implementation" is the common mantra
- Interfaces need to represent a crisp containment of a concept
  - But they should also include explicit mention of the essential conceptual dependencies — to encapsulate does not mean to hide everything

The Singleton Pattern

- Singleton is both a whisky and a pattern
  - The former is often better for design than the latter
- Half of its problems are the misconceptions developers hold about it
  - The other half is the publicity it receives in terms of how to apply and fix it
  - And the remaining half are in the Gang of Four book

> ***How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead.***
> **Kent Beck**

Global Domination

- Globals, Singletons and Monostate objects can mask significant, underlying dependencies
  - These approaches increase system coupling, reduce flexibility and complicate unit testing
  - And they further complicate the implementation in the presence of events, interrupts and threads
- Singleton has spawned a secondary industry of clever techniques targeting its deficiencies
  - However, when you find yourself at the bottom of a hole, stop digging...

Parameterize with Parameters

- Pass in configuration and behavioural parameters rather than having them global
  - Communicate through constructor or function arguments or template parameters, as appropriate
- Callout interfaces define the dependencies of each part that could be configured
  - E.g. the Strategy, Mock Object, Microkernel and Plug-in patterns
- Program to an interface, not an instance

The Selfish Object

- Instead of focusing on what an object can use or even be given, focus on what the object wants
  - In essence, express external dependencies by defining specific, narrow, plug-in-style interfaces
- Such self-centredness leads to a highly localised, open and testable architectural style
  - A result of consistent parameterization from above, which in the large — across subsystems and layers — results in "inversion layers" where the dependency horizon is nearby rather than distant

Hierarchical (Nested) Ownership

- Where possible, same scope should be responsible for acquisition and release
  - "Same scope" implies both block and member scope — empowered manager objects or separate strict custodian objects, e.g. *std::auto_ptr*
  - Other techniques can be used to support strict management, e.g. interface classes and non-*public* destructors enforce manager-based custody
  - Ownership is clear, code is simple, bugs are less likely

Shared Ownership

- Where shared ownership makes sense, it is best to consider a managed approach
  - Not the anarchy of "pass by guesswork" and "bad pointer days"
- In C++ this typically equates to the use of reference counting
  - Where a hierarchical and strict ownership scheme is not possible, shared ownership with tracking is clearer and less complex than an ad hoc transfer-and-hope-based scheme

Caveat Programmer

- But consider whether shared ownership is strictly necessary and not just a filler or crutch
  - Sometimes it results from vague ownership or as a way of dodging the question, rather than as a genuine need or useful point of flexibility
  - Reference counting can be a source of complexity, subtlety and pessimisation
- Hierarchical ownership should normally be preferred for its simplicity and clarity
  - There are more opportunities than are often realised

Smart Pointers or Smart Design?

- Smart pointers can sometimes be a useful aid in encapsulating management of indirection
  - Often, but not necessarily, managing object lifetimes
- But be restrained rather than enthusiastic in introducing smart pointers into interfaces
  - It is easy to create "clever" but obscure idiolects
  - Interfaces become coupled to the smart pointer code
  - Smart pointers can reduce rather than increase opportunities for optimisation and the creation of alternative implementations

Control Flow

> *Never offend people with style when you can offend them with substance.*
>
> **Sam Brown**

Jumpy Code

- Sometimes people are too ready to throw out the idea that code blocks should be modular
  - E.g. using multiple *return* points from a function
  - E.g. using *break* in loops rather than defining a loop's exit condition as the loop condition
  - E.g. using *continue* in loops instead of *if*
  - E.g. using *goto*
- Consider alternative arrangements before having a *break*

Muddled Code You Wouldn't Write

Muddled...

```
Node *node = this;
while(node)
{
    if(dynamic_cast<Group *>(node))
        return node;
    node = 0;
}
return 0;
```

```
next:
     p = mem[w];
     if(p == 0)
     {
          w = pop();
          goto next;
     }
```

Unmuddled...

```
return dynamic_cast<Group *>(this);
```

```
while(mem[w] == 0)
     w = pop();
```


Reasonable Code You Might Write

Breaking...

```
while(in_range(value))
{
    if(value == end)
        break;
    ...
}
```

```
for(int i = 0; i < len; ++i)
    if(name[i])
        name[i] = toupper(name[i]);
    else
        break;
```

Non-breaking...

```
while(in_range(value) && value != end)
{
    ...
}
```

```
for(int i = 0; i != len && name[i]; ++i)
    name[i] = toupper(name[i]);
```

The Structure of Code

- Code may be viewed linearly...
  - A sequential narrative that follows the path of software execution
  - Such a view can work in the small, but scales poorly, often with correspondingly one-dimensional logic
- Or hierarchically
  - The use of block structure to group and separate responsibilities and actions
  - This view is the one highlighted by indentation and folding/outlining editor views

Two Views of *realloc*

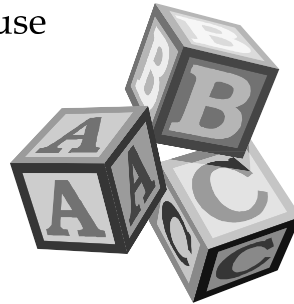
```
void *realloc(void *ptr, size_t new_size)
{
    if(!ptr)
    {
        return malloc(new_size);
    }
    if(new_size == 0)
    {
        free(ptr);
        return 0;
    }
    if(ptr == _Resize(ptr, new_size) &&
       _Size(ptr) >= new_size)
    {
        return ptr;
    }

    void *ptr_ptr = malloc(new_size);
    if(!ptr_new)
    {
        return 0;
    }

    memcpy(
        ptr_new, ptr,
        std::min(_size(ptr), new_size));
    free(ptr);
    return ptr_new;
}
```

```
void *realloc(void *ptr, size_t new_size)
{
    if(!ptr)
    {
        ptr = malloc(new_size);
    }
    else if(new_size == 0)
    {
        free(ptr);
        ptr = 0;
    }
    else if(ptr != _Resize(ptr, new_size) ||
            _Size(ptr) < new_size)
    {
        if(void *ptr_new = malloc(new_size))
        {
            memcpy(
                ptr_new, ptr,
                std::min(_size(ptr), new_size));
            free(ptr);
            ptr = ptr_new
        }
    }
    return ptr;
}
```

realloc *is an uncohesive function with three distinct behaviours — the behavioural branches are more visible on the right-hand side.*

Exceptions Are Modular

- Discontinuous control flow generally presents an obstacle to simple refactoring
  - How do you *break* across a method-call boundary? Or *return* across two method calls?
- Exceptions are different because they preserve block structure
  - They are invariant with respect to refactoring
  - They are exactly unlike *goto*



Careful with that *assert*, Eugene

- *assert* is a simple and useful mechanism for expressing functional truths
  - Can be used to check strict preconditions
  - Can be used to write unit functional tests simply
- However, very few people use it both effectively and consistently
  - E.g. asserting on expressions that are environment dependent, have side effects or are valid errors
  - Leads to source code pollution and fear of *NDEBUG*

Test the Right Thing

- *assert* can be used to express some contracts
  - But not all — there is more to the contract metaphor than assertions and assertability
- In practice, it can be worth expressing some preconditions and some data invariants
  - But it is rarely worth checking postconditions within a function — use external unit tests instead
  - As an example, consider the complexity of the postcondition of *std::sort*

In Closing

> *The only thing to do with good advice is to pass it on. It is never any use to oneself.*
>
> **Oscar Wilde**

Practice in Practice

- Part of C++'s challenge is in understanding its (many varied and subtle) mechanisms...
  - Of which there are more than enough to distract or detract from other programming goals
- And in part in finding the simplifying assumptions and styles that marshal them
  - So instead of focusing solely on mechanisms or unquestioned habits, programming can focus more on intent and reasoned practice