Reference Accounting

Patterns for Managing Non-nested Object Lifetimes

Kevlin Henney

Curbralan Ltd

kevlin@curbralan.com

C++ CONNECTIONS

Agenda

- Intent
  - Present a structured design vocabulary for working with reference tracking solutions for object-lifetime management
- Content
  - Overview of pattern concepts
  - Overview of *Reference Accounting*
  - *Reference Accounting* core patterns

C++
CONNECTIONS

Overview of Pattern Concepts

- Intent
  - Present core pattern terminology and ideas
- Content
  - Patterns and pattern quality
  - Pattern communities
  - Pattern stories and sequences
  - Pattern compounds
  - Pattern languages

C++ CONNECTIONS

Patterns

- A pattern documents a recurring problem–solution pairing within a given context
  - A pattern is more than either the problem or the solution structure
  - A pattern contributes to design vocabulary
- The full form for a pattern should emphasise forces and consequences
  - Also stating clearly the essential problem and solution — these are often missing

C++ CONNECTIONS

Pattern Quality

- Contrary to popular belief, a pattern is not by definition "good"
  - There are also poor patterns — dysfunctional designs recur, through either habit or fashion
  - There are poor applications of good patterns
- A poor pattern or pattern application can be characterised as being out of balance
  - Its consequences and forces do not adequately match up

C++ CONNECTIONS

Patterns of Misunderstanding

- There are misconceptions concerning the pattern concept worth clearing up...
  - *Design Patterns* is a limited subset of design patterns and the pattern concept
  - Patterns are not frameworks, components, blueprints or parameter-based collaborations
  - Patterns are more than just a sample class diagram of the solution
  - Patterns may be language dependent

C++ CONNECTIONS

Pattern Communities

- Patterns can be used in isolation with some degree of success
  - Foci for discussion or point solutions
  - Offer localised design ideas
- However, patterns are, in truth, gregarious
  - They're fond of the company of patterns
  - Interwoven nature of design means patterns inevitably enlist others for expression and variation, competition and cooperation

C++
CONNECTIONS

Pattern Stories and Sequences

- A pattern story brings out the sequence of patterns applied in a given design example
  - They capture the conceptual narrative behind a given piece of design, real or illustrative
  - Forces and consequences played out in order
- More generally, pattern sequences are specific ordered applications of patterns
  - A pattern story is to a pattern sequence as a pattern example is to a single pattern

C++ CONNECTIONS

Pattern Compounds

- Pattern compounds capture commonly recurring subcommunities of patterns
  - Most patterns are compound, at one level or another or from one point of view or other
- We can see many pattern compounds as named pattern subsequences
  - They are commonly recurring fragments that can be further decomposed, if desired

C++ CONNECTIONS

Pattern Languages

- A pattern language connects many patterns to capture a broader set of paths
  - The intent of a language is to generate a particular kind of system or subsystem
  - A pattern language can describe idiomatic design style
- There may be many possible and practical sequences through a pattern language
  - A sequence is a narrow language

C++ CONNECTIONS

Overview of *Reference Accounting*

- Intent
  - Summarise general motivation, structure and consequences of *Reference Accounting* pattern language
- Content
  - Hierarchical versus shared ownership
  - *Reference Accounting* patterns
  - *Reference Accounting* benefits and liabilities

C++ CONNECTIONS

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 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 custody
  - Ownership is clear, bugs are less likely

C++ CONNECTIONS

Shared Ownership

- Where shared ownership makes sense, it is best to consider a managed approach
  - Not the anarchy of ad hoc ownership transfer
  - Garbage collection or reference counting
- A number of roles in a counted approach...
  - The *shared object* is the *body* to be managed
  - The body pointer (or wrapper) is the *handle*
  - The *count* represents is the count of the number of handles referring to a body

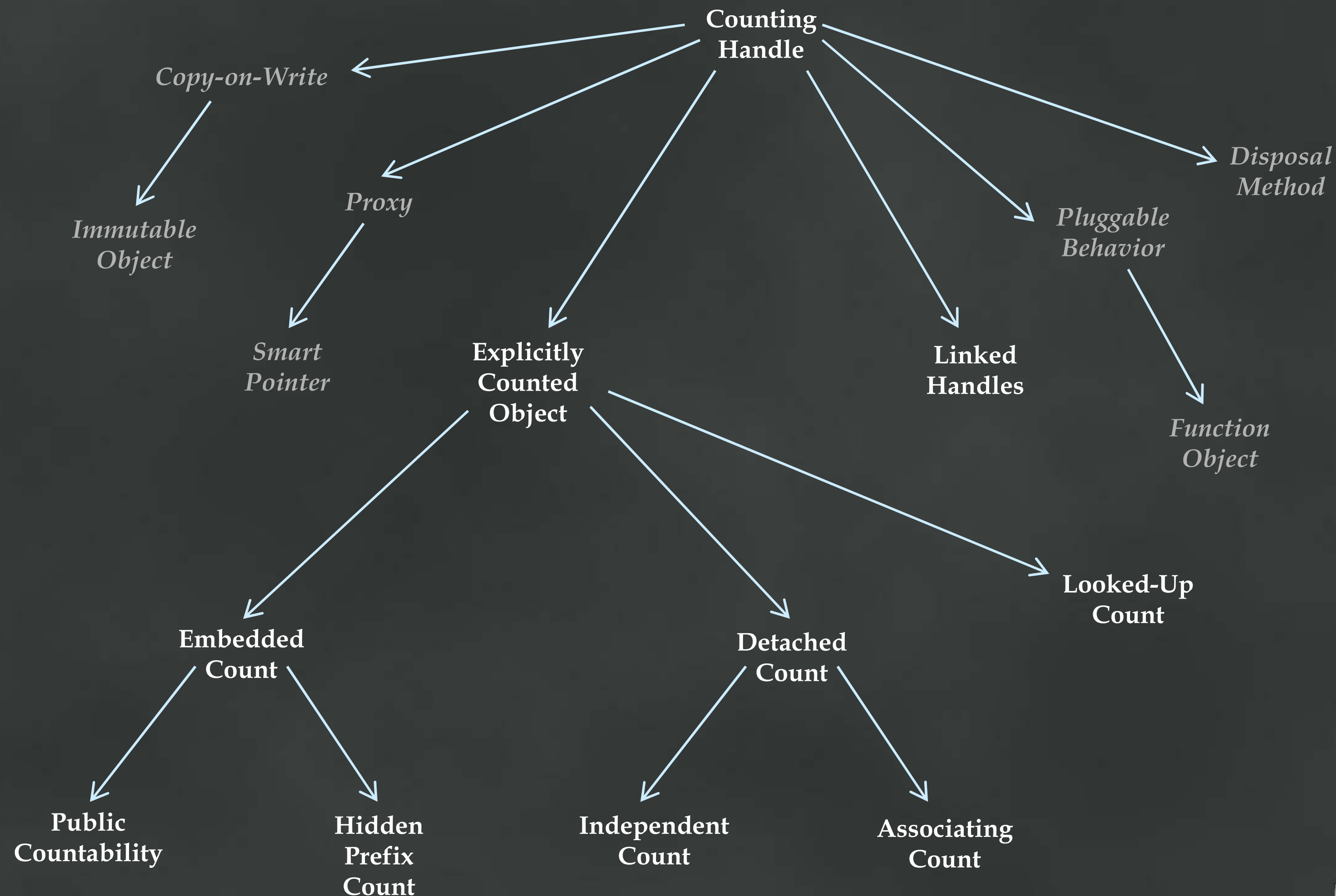
C++ CONNECTIONS

Count and Body Configuration

- What is the physical relationship between the count and the body?
  - Count and body may be attached or detached
  - Count may be external to handle–body
- How intrusive is reference counting with respect to the counted body?
  - Countability may be a property of the body or added and managed by the handle
  - Countability may be deduced or explicit

C++ CONNECTIONS

Reference Accounting Patterns



C++ CONNECTIONS

Reference Accounting Benefits

- Representation sharing...
  - Avoid the cost of repeated object copying by sharing representation and tracking it
- Simplicity...
  - 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
  - Deterministic object lifetimes

C++ CONNECTIONS

Reference Accounting Liabilities

- Representation sharing...
  - Proxy rather than direct access is required to support Copy-on-Write or, alternatively, just use Immutable Objects
  - Representation sharing is not always an optimisation, and there may be better alternatives, e.g. short-string optimisation
- Concurrency...
  - Thread safety of tracking mechanism

C++ CONNECTIONS

Reference Accounting Liabilities

- Cycles...
  - Cyclic dependencies leading to orphaned rings of objects are possible with Proxies
  - Either weak or hierarchical ownership can break cycles
- Complexity...
  - Code can become more complex, and not always necessarily

C++ CONNECTIONS

Reference Accounting Core Patterns

- Intent
  - Present descriptions for the core patterns in the Reference Accounting language
- Content
  - Counted Handle, Explicitly Counted Object, Embedded Count, Public Countability, Hidden Prefix Count, Detached Count, Independent Count, Associating Count, Looked-Up Count, Linked Handles

C++ CONNECTIONS

Counting Handle Pattern

- How can a shared object allocated have its lifetime managed transparently?
  - Simple access for heap-allocated object, but without explicit intervention from its users
- Introduce or nominate a handle object through which the shared object is used
  - The handle encapsulates the responsibility for tracking references to the shared object and for its deletion

C++ CONNECTIONS

Visibility of *Counting Handle* Role

- Hidden...
  - Separation of a single value-based object into a handle–body pair
  - Modification of shared representation leads to Copy-on-Write or use of Immutable Objects
- Explicit...
  - Use of a Proxy, typically a Smart Pointer, to access a known shared object

C++ CONNECTIONS

Explicitly Counted Object Pattern

- How can a Counting Handle know that it is the last one to refer to a shared object?
  - When the count falls to zero, the shared object can be disposed of
- Introduce an explicit count to track number of references to a shared object
  - The count is incremented and decremented by the Counting Handles as they reference and unreference a shared object

C++ CONNECTIONS

Embedded Count Pattern

- How can an Explicitly Counted Object, or its users, be aware of its counting status?
  - This may be needed by the object itself or by code pointing to the shared object that has no access to a corresponding Counting Handle
- Embed the reference count in the Explicitly Counted Object itself
  - No additional allocation needed for the count
  - But cannot support weak referencing

C++ CONNECTIONS

Public Countability Pattern

- How can users of a shared object, with an Embedded Count, and the object itself be aware of the usage count?
  - The object needs autonomy and awareness
- Make the reference counting capability an explicit part of the type
  - This means that both the count's usage interface and representation are part of the shared object's class and design intent

C++ CONNECTIONS

Public Countability Sketch

```
template<typename countable_type>
class counting_ptr
{
public:
    explicit counting_ptr(countable_type *);
    countable_ptr(const counting_ptr &);
    ~counting_ptr();
    counting_ptr &operator=(const counting_ptr &);
    ...
private:
    countable_type *body;
};
```

```
class countable_type
{
public:
    void acquire() const;
    bool release() const;
    bool acquired() const;
    ...
    mutable size_t count;
};
```

The target body class must support a countable interface and define a count data member

C++ CONNECTIONS

Hidden Prefix Count Pattern

- How can an Explicitly Counted Object be type independent of an Embedded Count?
  - While still ensuring that the count is integral to the shared object
- Provide additional memory for the count in the memory preceding the shared object
  - The count is a property of the object's memory — and its allocation — not of its type

C++ CONNECTIONS

Hidden Prefix Count Sketch

```
// counting_ptr class template definition as before
struct countable_new;
extern const countable_new countable;
void *operator new(size_t, const countable_new &);
void operator delete(void *, const countable_new &);
```

Overloaded operator new *prefixes allocated body with counting header and* counting_ptr *is written to use helper functions that manipulate this header*

```
struct countable_new_header { size_t count; };
void acquire(const void *);
bool release(const void *);
bool acquired(const void *);
```

The target body type is independent of counting, but countable instances must be allocated using the overloaded operator new

```
counting_ptr<type> ptr = new(countable) type;
```

C++ CONNECTIONS

Detached Count Pattern

- How can an Explicitly Counted Object be fully independent of the reference count?
  - A need for type and instance independence
- Introduce a separate object managed by the Counting Handle to hold the count
  - The handle is responsible for lifetimes both of shared object and corresponding count object
  - No recovery of count from shared object
  - Easy to plug-in additional behaviours

C++ CONNECTIONS

Independent Count Pattern

- How can a Detached Count be fully independent of the shared object it tracks?
  - Two objects are managed, but minimal interaction and dereferencing
- Have the Counting Handle hold separate pointers to the Explicitly Counted Object and the Detached Count
  - Single level of indirection for shared object access, but two pointers in the handle

C++ CONNECTIONS

Independent Count Sketch

```
template<typename type>
class counting_ptr
{
public:
    explicit counting_ptr(type *);
    counting_ptr(const counting_ptr &);
    ~counting_ptr();
    counting_ptr &operator=(const counting_ptr &);
    ...
private:
    size_t *count;
    type   *body;
};
```

```
class type { ... };
```

```
typedef ... size_t;
```

Target body type is independent of counting, and is not physically related to the counter

C++ CONNECTIONS

Associating Count Pattern

- How can a Detached Count know and affect an Explicitly Counted Object?
  - Support operations that modify the shared object in a way that is uniformly visible to all of its Counting Handles
- Have the Detached Count hold a pointer to the Explicitly Counted Object
  - Simplest support for replacement of shared object and plug-in behaviours

C++ CONNECTIONS

Associating Count Sketch

```
template<typename type>
class counting_ptr
{
public:
    explicit counting_ptr(type *);
    counting_ptr(const counting_ptr &);
    ~counting_ptr();
    counting_ptr &operator=(const counting_ptr &);
    ...
private:
    counted_link *link;
};
```

```
struct counted_link
{
    size_t count;
    type   *body;
};
```

```
class type { ... };
```

Target body type is independent of counting, and is two levels of indirection removed from the handle via the count

C++ CONNECTIONS

Looked-Up Count Pattern

- How can Explicitly Counted Objects be grouped together without type intrusion?
  - Acted on collectively or have single instances replaced or disposed of before count zero
- Manage shared objects and their counts collectively in a separate managed object
  - Typically use some identity of the shared object, not just its address, as the key for its direct access from the Counting Handle

C++ CONNECTIONS

Looked-Up Count Sketch

```
template<typename target_type>
class tracking_ptr
{
    ...
    typename target_type::key_type   key;
    tracking_repository<target_type> *repository;
};
```

```
template<typename target_type>
class tracking_repository
{
    ...
    typedef typename type::key_type key_type;
    struct shared
    {
        size_t      count;
        target_type *target;
    };
    std::map<key_type, shared> tracked;
};
```

C++ CONNECTIONS

Linked Handles Pattern

- How can all Counting Handles associated with a shared object be acted on together?
  - But without introducing any intermediate objects, such as managers?
- Introduce bidirectional links between the Counting Handles
  - Each is aware of both the shared object and other Counting Handles to the same object
  - Not safe for sharing across threads

C++ CONNECTIONS

Linked Handles Sketch

```
template<typename type>
class linked_ptr
{
public:
    explicit linked_ptr(type *);
    linked_ptr(const linked_ptr &);
    ~linked_ptr();
    linked_ptr &operator=(const linked_ptr &);
    ...
private:
    type        *body;
    linked_ptr *next, *previous;
};
```

```
class type { ... };
```

Target body type is independent of counting, and there is no explicit count

C++ CONNECTIONS

In Conclusion

- Reference counting can be a good fit for some object ownership models
  - Smart pointers offer a suitable persona for managing publicly shared objects
  - Other handle forms are suitable for more private uses of sharing
- There are many ways to track references
  - There is no one-size-fits-all solution
- But...

C++ CONNECTIONS

Caveat Programmer: Shared Custody

- Consider whether shared ownership is strictly necessary
  - 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

C++ CONNECTIONS

Caveat Programmer: Smart Pointers

- Be restrained rather than enthusiastic in introducing smart pointers into interfaces
  - It is easy to create "clever" but obscure idiolects that act as barriers to understanding
  - The interface is now coupled to the smart pointer code
  - Smart pointers can reduce rather than increase opportunities for optimisation and the creation of alternative implementations

C++ CONNECTIONS