Smart Pointers in C++

CS20006: Software Engineering

Lecture 49

Prof. Partha Pratim Das

भारतीय प्रौद्योगिकी संस्थान खड़गपुर
Indian Institute of Technology
Kharagpur

April 2016

Motivation

- Imbibe a culture to write "good" C++ code
  - Correct: Achieves the functionality
  - Bug free: Free of programming errors
  - Maintainable: Easy to develop & support
  - High performance: Fast, Low on memory.

> *"C++ is an abomination to society, and is doubtlessly responsible for hundreds of millions of lost hours of productivity."*
>
> – Space Monkey as posted on kuro5him.org

Agenda

- Raw Pointers – A recap
  - Operations
  - Consequences of not being an FCO
  - Pointer Hazards
- A Pointer-free World
  - Pointers vis-à-vis Reference
  - Quick Tour of Pointer-Free Languages

Agenda

- Smart Pointers in C++
  - Policies
    - Storage
    - Ownership
    - Conversion
      - Implicit Conversions
      - Null Tests
  - Checking
  - Other Design Issues

> "Understanding pointers in C is not a skill, it's an aptitude…"
>
> – Joel Spolsky in "Joel on Software - The Guerrilla Guide to Interviewing"

A Raw Deal?

RAW POINTERS

What is a Raw Pointer?

- Raw Pointer Operations
  - Dynamic Allocation (result of) or `operator&`
  - Deallocation (called on)
  - De-referencing `operator*`
  - Indirection `operator->`
  - Assignment `operator=`
  - Null Test `operator!` (`operator== 0`)
  - Comparison `operator==`, `operator!=`, …
  - Cast `operator(int)`, `operator(T*)`
  - Address Of `operator&`
  - Address Arithmetic `operator+`, `operator-`, `operator++`, `operator--`, `operator+=`, `operator-=`
  - Indexing (array) `operator[]`

What is a Raw Pointer?

- Typical use of Pointers
  - Essential – Link (‘next’) in a data structure
  - Inessential – Apparent programming ease
    - Passing Objects in functions: void MyFunc(MyClass *);
    - ‘Smart’ expressions: while (p) cout << *p++;
- Is not a “First Class Object”
  - An integer value is a FCO
- Does not have a “Value Semantics”
  - Cannot COPY or ASSIGN at will
- Weak Semantics for “Ownership” of pointee

Ownership Issue of Pointers

- Ownership Issue – ASSIGN problem

```
// Create ownership
MyClass *p = new MyClass;

// Lose ownership
p = 0;
```

- Memory Leaks!

Ownership Issue of Pointers

- Ownership Issue – COPY problem

```
// Create ownership
MyClass *p = new MyClass;
// Copy ownership – no Copy Constructor!
MyClass *q = p;
// Delete Object & Remove ownership
delete q;
// Delete Object – where is the ownership?
delete p;
```

- Double Deletion Error!

Ownership Issue of Pointers

- Ownership Issue – SCOPE problem

```
void MyAction() {
        // Create ownership
        MyClass *p = new MyClass;
        // What if an exception is thrown here?
        p->Function();
        // Delete Object & Remove ownership
        delete p;
}
```

- Memory Leaks due to stack unrolling!

Ownership Issue of Pointers

```
void MyAction() {
        MyClass *p = 0;
        try {
                MyClass *p = new MyClass;
                p->Function();
        }
        catch (…) {
                delete p; // Repeated code
                throw;
        }
        delete p;
}
```

- try-catch solves this case

Ownership Issue of Pointers

```
void MyDoubleAction() {
        MyClass *p = 0, *q = 0;
        try {
                MyClass *p = new MyClass;
                p->Function();
                MyClass *q = new MyClass;
                q->Function();
        } catch (…) {
                delete p; // Repeated code
                delete q; // Repeated code
                throw;
        }
        delete p;
        delete q;
}
```

- Exceptional path dominates regular path

Pointer Hazards

- Pointer issues dominate all Memory Errors in C++
  - Null Pointer Dereference
  - Dangling pointers
  - Double Deletion Error
  - Allocation failures
  - Un-initialized Memory Read
  - Memory Leaks
  - Memory Access Errors
  - Memory Overrun
  - <u>*Exceptiona*</u>l Hazards

> *"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."*
>
> – Weinberg's Second Law

Reality or Utopia?

A POINTER-FREE WORLD

How to deal with an Object?

- The object itself –
  - *by value*
    - Performance Issue
    - Redundancy Issue
- As the memory address of the object –
  - *by pointer*
    - Lifetime Management Issue
    - Code Prone to Memory Errors
- With an alias to the object –
  - *by reference*
    - Good when null-ness is not needed
    - Const-ness is often useful

Pointers vis-à-vis Reference

- Use ‘Reference’ to Objects when
  - Null reference is not needed
  - Reference once created does not need to change
- Avoids
  - The security problems implicit with pointers
  - The (pain of) low level memory management (i.e. delete)
- W/o pointer – Use
  - Garbage Collection

> *“Avoid working with pointers.*
> *Consider using references instead.”*
> “Avoiding Common Memory Problems in C++” – MSDN Article

The Smartness …

SMART POINTERS IN C++

What is Smart Pointer?

- A Smart pointer is a C++ object
- Stores pointers to dynamically allocated (heap / free store) objects
- Improves raw pointers by implementing
  - Construction & Destruction
  - Copying & Assignment
  - Dereferencing:
    - `operator->`
    - unary `operator*`
- *Grossly* mimics raw pointer syntax & semantics

What is Smart Pointer?

- Performs extremely useful support tasks
  - RAII – Resource Acquisition is Initialization Idiom
  - Selectively disallows “unwanted” operations
    - Address Arithmetic
  - Lifetime Management
    - Automatically deletes dynamically created objects at appropriate time
    - On face of exceptions – ensures proper destruction of dynamically created objects
    - Keeps track of dynamically allocated objects shared by multiple owners
  - Concurrency Control

A Simple Smart Pointer

```
template <class T> class SmartPtr {
public:
    // Constructible. No implicit conversion from Raw ptr
    explicit SmartPtr(T* pointee): pointee_(pointee);
    // Copy Constructible
    SmartPtr(const SmartPtr& other);
    // Assignable
    SmartPtr& operator=(const SmartPtr& other);
    // Destroys the pointee
    ~SmartPtr();
    // Dereferencing
    T& operator*() const { ... return *pointee_; }
    // Indirection
    T* operator->() const { ... return pointee_; }
private:
    T* pointee_; // Holding the pointee
};
```

A Smart Pointer mimics a Raw Pointer

```
class MyClass {
public:
   void Function();
};

// Create a smart pointer as an object
SmartPtr<MyClass> sp(new MyClass);

// As if indirecting the raw pointer
sp->Function(); // (sp.operator->())->Function()

// As if dereferencing the raw pointer
(*sp).Function();
```

The Smartness ...

- It always points either to a valid allocated object or is NULL.
- It deletes the object once there are no more references to it.
- Fast. Preferably zero de-referencing and minimal manipulation overhead.
- Raw pointers to be only explicitly converted into smart pointers. Easy search using grep is needed (it is unsafe).
- It can be used with existing code.

The Smartness …

- Programs that don’t do low-level stuff can be written exclusively using this pointer. No Raw pointers needed.
- Thread-safe.
- Exception safe.
- It shouldn’t have problems with circular references.

Storage Policy

SMART POINTERS IN C++

3–Way Storage Policy

- The Storage Type (T*)
  - The type of pointee.
    - Specialized pointer types possible: FAR, NEAR.
  - By “default” – it is a raw pointer.
    - Other Smart Pointers possible – When layered
- The Pointer Type (T*)
  - The type returned by `operator->`
    - Can be different from the storage type if proxy objects are used.
- The Reference Type (T&)
  - The type returned by `operator*`

Ownership Management Policy

SMART POINTERS IN C++

Ownership Management Policy

- Smart pointers are about ownership of pointees
- Exclusive Ownership
  - Every smart pointer has an exclusive ownership of the pointee
  - Destructive Copy
    - std::unique_ptr
- Shared Ownership
  - Ownership of the pointee is shared between Smart pointers
  - std::shared_ptr
  - std::weak_ptr
  - Track the Smart pointer references for lifetime
    - Reference Counting
    - Reference Linking

Ownership Policy: Destructive Copy

- Exclusive Ownership Policy
- Transfer ownership on copy
- Source Smart Pointer in a copy is set to NULL
- Available in C++ Standard Library
  - std::unique_ptr
- Implemented in
  - Copy Constructor
  - operator=

Ownership Policy: Destructive Copy

```
template <class T> class SmartPtr {
public:
SmartPtr(SmartPtr& src) { // Src ptr is not const
        pointee_ = src.pointee_; // Copy
        src.pointee_ = 0; // Remove ownership for src ptr
}
SmartPtr& operator=(SmartPtr& src) { // Src ptr is not const
        if (this != &src) { // Check & skip self-copy
                delete pointee_; // Release destination object
                pointee_ = src.pointee_; // Assignment
                src.pointee_ = 0; // Remove ownership for src ptr
        }
        return *this; // Return the assigned Smart Pointer
} ...
};
```

Ownership Policy: Destructive Copy – The Maelstrom Effect

- Consider a call-by-value

```
void Display(SmartPtr<Something> sp); ...
SmartPtr<Something> sp(new Something);
Display(sp); // sinks sp
```

- Display acts like a maelstrom of smart pointers:
  - It sinks any smart pointer passed to it.
  - After Display(sp) is called, sp holds the null pointer.
- Lesson – Pass Smart Pointers by Reference.
- *Smart pointers with destructive copy cannot usually be stored in containers and in general must be handled with care.*

STL Containers need FCO.

Ownership Policy: Destructive Copy – Advantages

- Incurs almost no overhead.
- Good at enforcing ownership transfer semantics.
  - Use the “maelstrom effect” to ensure that the function takes over the passed-in pointer.
- Good as return values from functions.
  - The pointee object gets destroyed if the caller doesn't use the return value.
- Excellent as stack variables in functions that have multiple return paths.
- Available in the standard – std::auto_ptr.
  - Many programmers will get used to this behavior sooner or later.

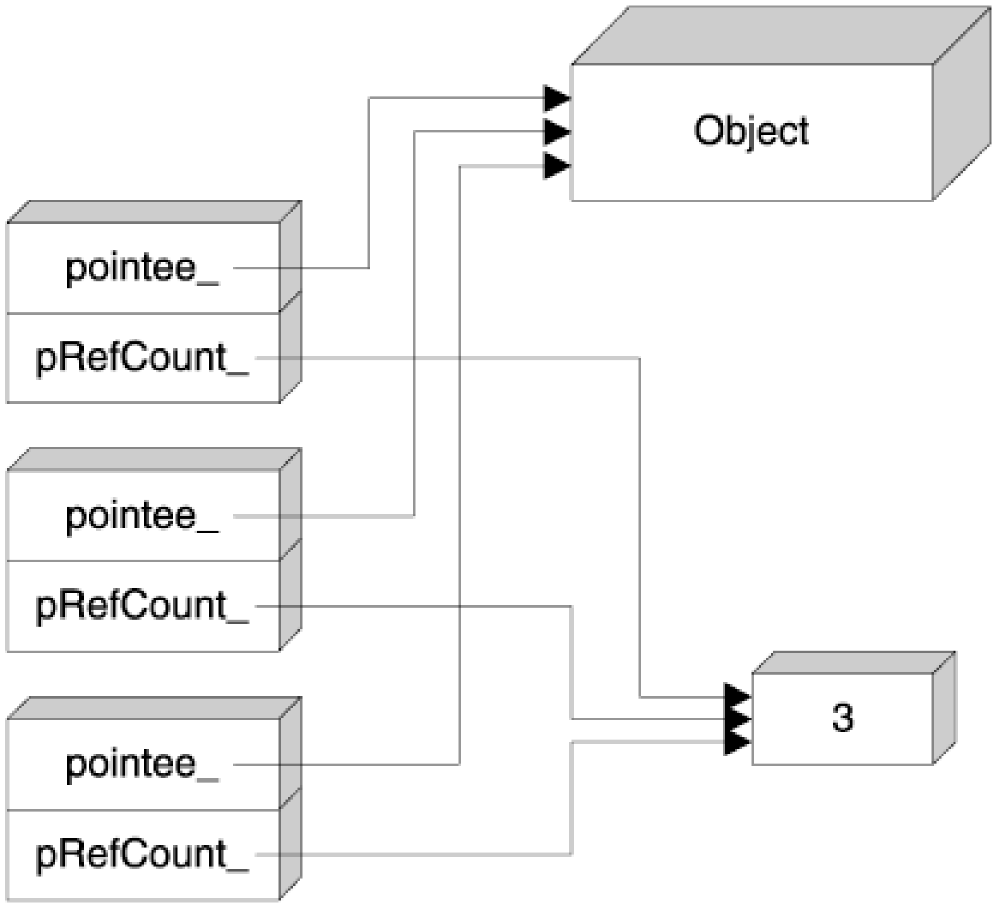
Ownership Policy:
Reference Counting

- Shared Ownership Policy
- Allow multiple Smart pointers to point to the same pointee
- A count of the number of Smart pointers (references) pointing to a pointee is maintained
- Destroy the pointee Object when the count equals 0
- Do not keep: raw pointers and smart pointers to the same object.

Ownership Policy:
Reference Counting

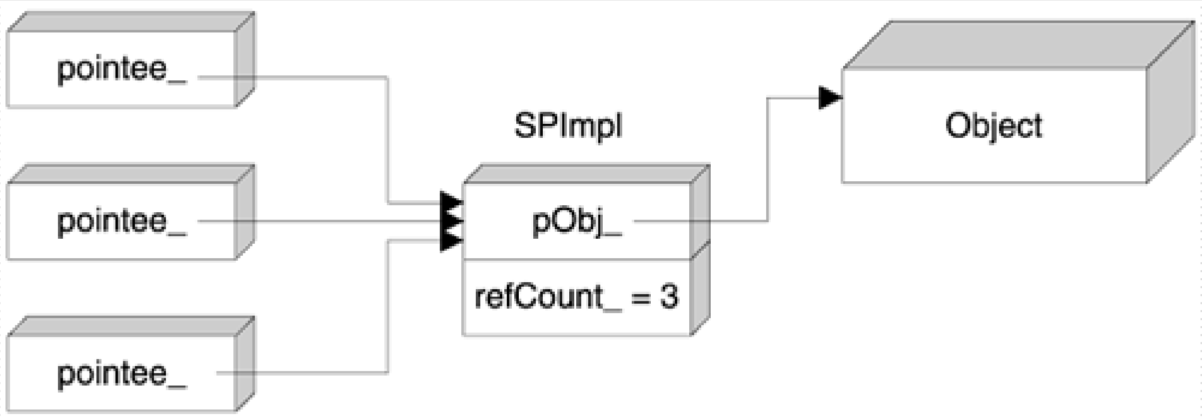
- Variant Sub-Policies include
  - Non-Intrusive Counter
    - Multiple Raw Pointers per pointee
    - Single Raw Pointer per pointee
  - Intrusive Counter
- Implemented in
  - Constructor
  - Copy Constructor
  - Destructor
  - `operator=`

Ownership Policy:
Reference Counting: Non-Intrusive Counter



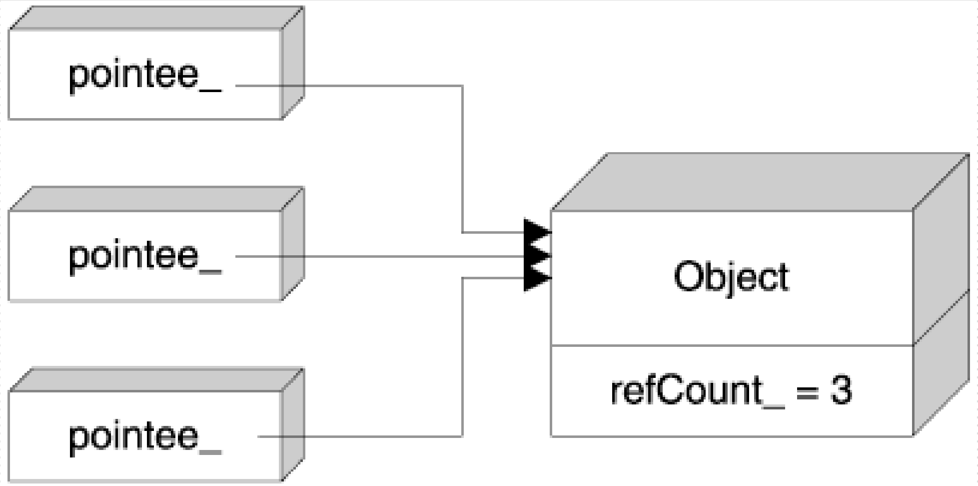
- Additional count pointer per Smart Pointer.
- Count in Free Store
- Allocation of Count may be slow & wasteful because it is too small

Ownership Policy: Reference Counting: Non-Intrusive Counter



- Additional count pointer removed.
- But additional access level means slower speed.

Ownership Policy:
Reference Counting: Intrusive Counter

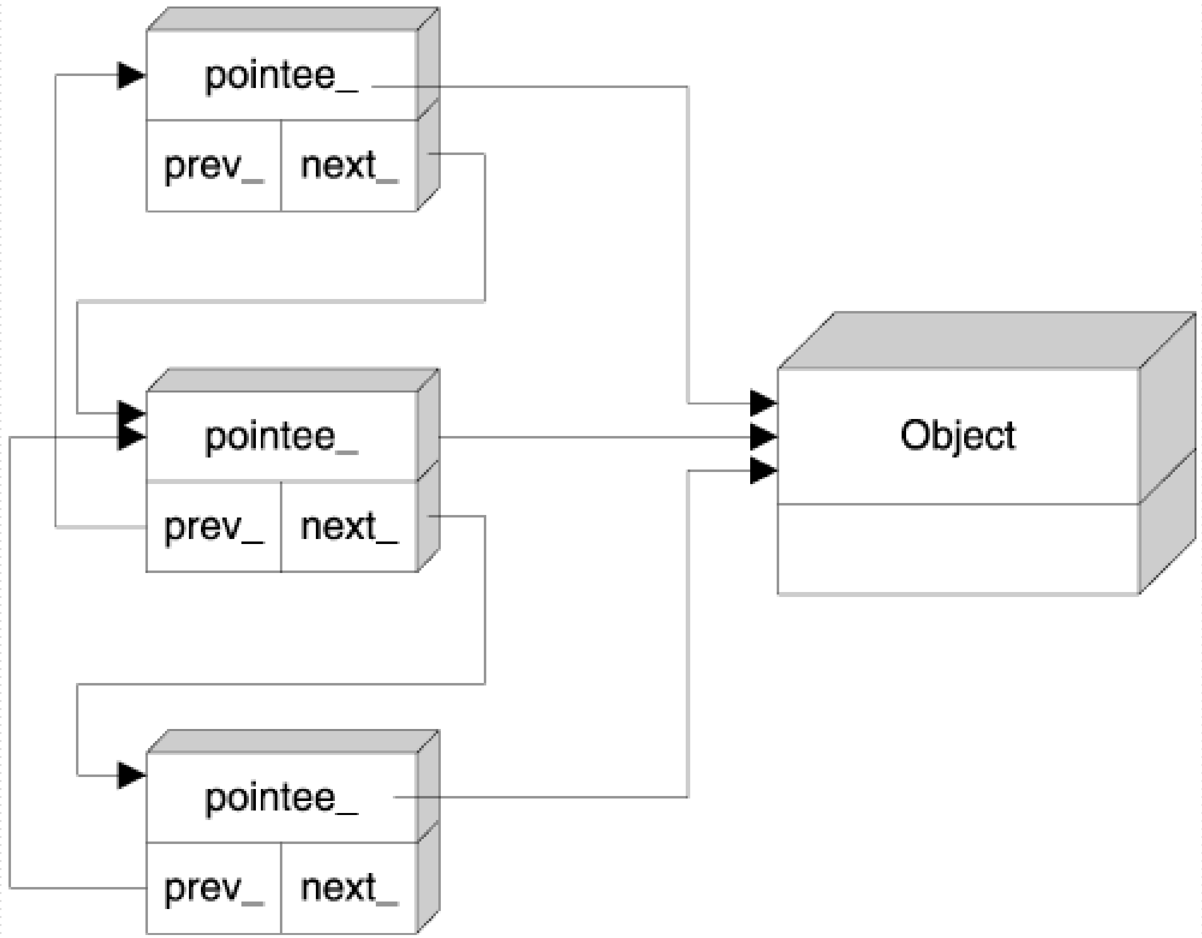


- Most optimized RC Smart Pointer
- Cannot work for an already existing design
- Used in COM

Ownership Policy:
Reference Linking

- Shared Ownership Policy
- Allow multiple Smart pointers to point to the same pointee
- All Smart pointers to a pointee are linked on a chain
  - The exact count is not maintained – only check if the chain is null
- Destroy the pointee Object when the chain gets empty
- Do not keep: raw pointers and smart pointers to the same object.

Ownership Policy: Reference Linking

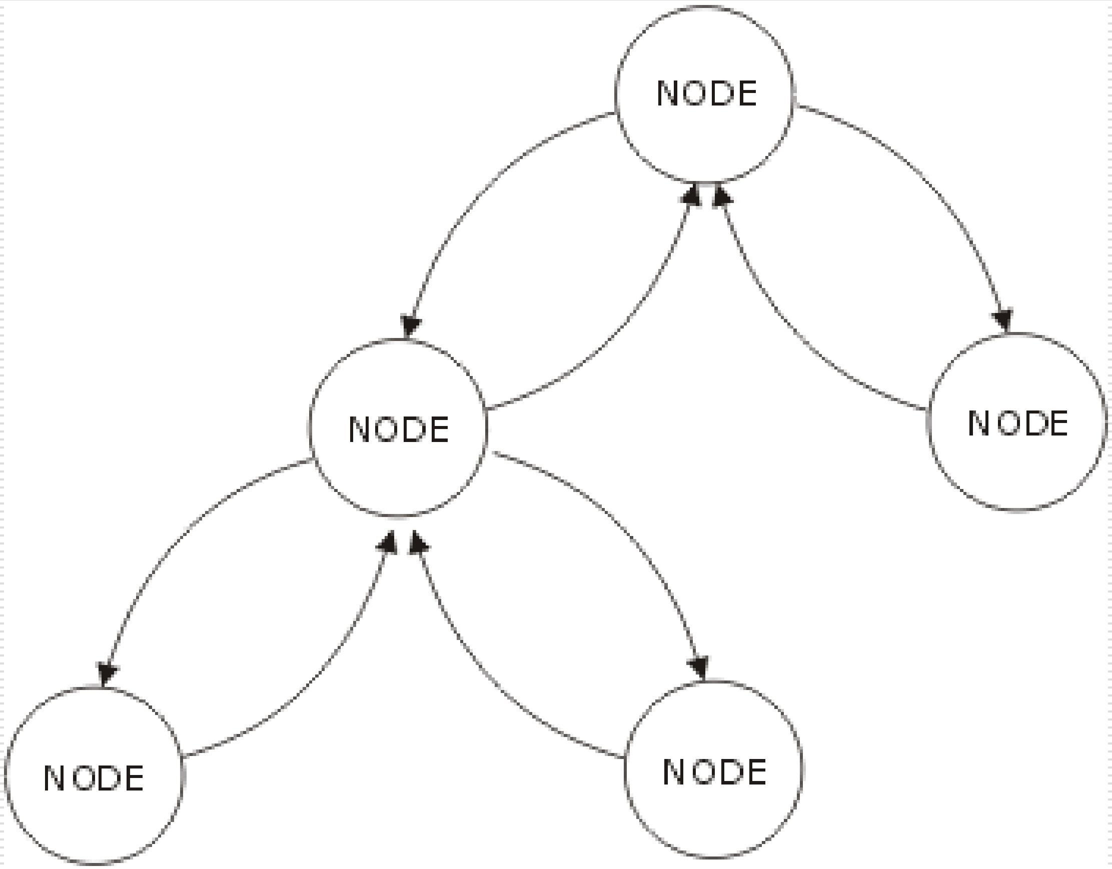


- Overhead of 2 additional pointers
- Doubly-linked list for constant time:
  - Append,
  - Remove & Empty detection.

Ownership Policy: Reference Management – Disadvantage

- Circular / Cyclic Reference
  - Object A holds a smart pointer to an object B. Object B holds a smart pointer to A. Forms a cyclic reference.
    - Typical for a Tree: Child & Parent pointers
  - Cyclic references go undetected
    - Both the two objects remain allocated forever
    - Resource Leak occurs.
  - The cycles can span multiple objects.

Ownership Policy: Cyclic Reference – Hack



The Hack

- Use Smart pointer (std::shared_ptr) from Parent to Child.
  - "Data Structure" Pointers
- Use Weak pointer (std::weak_ptr) from Child to Parent.
  - "Algorithm" Pointers

Ownership Policy:
Cyclic Reference – Solution

- Maintain two flavors of RC Smart Pointers
  - “Strong” pointers that really link up the data structure (Child / Sibling Links). They behave like regular RC. std::shared_ptr
  - “Weak” pointer for cross / back references in the data structure (Parent / Reverse Sibling Links). std::weak_ptr
- Keep two reference counts:
  - One for total number of pointers, and
  - One for strong pointers.
- While dereferencing a weak pointer, check the strong reference count.
  - If it is zero, return NULL. As if, the object is gone.

Implicit Conversion

SMART POINTERS IN C++

Implicit Conversion

- Consider

```
// For maximum compatibility this should work
void Fun(Something* p); ...
SmartPtr<Something> sp(new Something);
Fun(sp); // OK or error?
```

- User-Defined Conversion (cast)

```
template <class T> class SmartPtr {
public:
        operator T*() // user-defined conversion to T*
        { return pointee_; } ...
};
```

- User-unattended access to the raw pointer can defeat the purpose of the smart pointer

Implicit Conversion: The Pitfall

- This compiles okay!!!

```
// A gross semantic error that goes undetected at compile time
SmartPtr<Something> sp; ...
delete sp; // Compiler passes this by casting to raw pointer
```

- Ambiguity Injection solves ...

```
template <class T> class SmartPtr {
public:
        operator T*() // User-defined conversion to T*
        { return pointee_; }
        operator void*() // Added conversion to void*
        { return pointee_; } ...
};
```

- Prefer Explicit Conversion over Implicit
  - Use GetImpl() & GetImplRef()

"When in doubt, use brute force." – Ken Thompson

Null Tests

SMART POINTERS IN C++

Null Tests

- Expect the following to work?

```
SmartPtr<Something> sp1, sp2;
Something* p; ...

if (sp1)        // Test 1: direct test for non-null pointer ...
if (!sp1)       // Test 2: direct test for null pointer ...
if (sp1 == 0)   // Test 3: explicit test for null pointer ...
```

- Implicit conversion to:
  - T*
  - void *
- Implicit conversion → Risky delete → Ambiguity Injection → Ambiguity causes compilation failures

Null Tests

- Overload operator!

```
template <class T> class SmartPtr {
public:
        // Returns true iff pointee is NULL
        bool operator!()
        { return pointee_ == 0; }
};

if (sp1)        // Rewrite as if (!!sp1)
if (!sp1)       // Works fine
if (sp1 == 0)   // Does not work
```

Checking Policy

SMART POINTERS IN C++

Checking Policy

- Applications need various degrees of safety:
  - Computation-intensive – optimize for speed.
  - I/O intensive –allows better runtime checking.
- Two common models:
  - Low safety / High speed (critical areas).
  - High safety / Lower speed.
- Checking policy with smart pointers:
  - Checking Functions
    - Initialization Checking &
    - Checking before Dereferencing
  - Error Reporting

Checking Policy: Initialization Checking

- Prohibit a Smart Pointer from being NULL
  - A Smart Pointer is always valid
  - Loses the ‘not-a-valid-pointer’ idiom
  - How would default constructor initialize raw pointer?

```
template <class T> class SmartPtr {
public:
        // Prohibit NULL for initialization
        SmartPtr(T* p): pointee_(p) {
                if (!p) throw NullPointerException();
        } ...
};
```

Checking Policy:
Checking before Dereferencing

- Dereferencing a null pointer is undefined!

```
if (p)   { /* Dereference & use p */ }
else     { /* Handle null pointer condition */ }
```

- Implemented in
  - Operator->
  - Operator*

```
template <class T> class SmartPtr {
public:
        T& operator*() const { // Dereferencing
                if (!pointee_) throw NullPointerException();
                else return *pointee_; }
        T* operator->() const { // Indirection
                if (!pointee_) throw NullPointerException();
                else return pointee_; }
 ...
};
```

Checking Policy: Error Reporting

- Throw an exception to report an error
- Use ASSERT in debug build
  - Checking (debug) + Speed (release)
- Lazy Initialization – construct when needed
  - Operator->
  - Operator*

```
template <class T> class SmartPtr {
public:
        T& operator*() { // Dereferencing. No Constant
                if (!pointee_) pointee_ = new T;
                return *pointee_; }
        T* operator->() { // Indirection. No Constant
                if (!pointee_) pointee_ = new T;
                return pointee_; }
 ...
};
```

Other Design Issues

SMART POINTERS IN C++

Other Design Issues

- Comparison of two Smart Pointers
  - Equality, Inequality, Ordering
- Checking and Error Reporting
  - Initialization checking
  - Checking before dereference
- const-ness
  - Smart Pointers to const and
  - const Smart Pointers
- Arrays
- Multi-Threading / Locks
  - Proxy Objects

References: Books

- **Effective C++ by** *Scott Meyers*
- **More Effective C++: 35 New Ways to Improve Your Programs and Designs** – *Scott Meyers*, Pearson Education & AWP 1999
- **Modern C++ Design: Generic Programming & Design Pattern Applied** – *Andrei Alexandrescu*, Pearson Education 2001
- **C++ Templates: The Complete Guide** – *David Vandevoorde & Nicolai M. Josuttis*, Pearson Education & AWP 2003
- **Exceptional C++ by** *Herb Sutter*
- **More Exceptional C++ by** *Herb Sutter*
- **The C++ Programming Language by** *Bjarne Stroustrup*

Thank You

Don't Beware of Pointers –

Just Be Aware of Smart Pointers