Italian C++ Community

www.italiancpp.org

(slightly) smarter

SMART PTRS

Carlo Pescio

sigeo
Servizi Informativi Geografici

ELETTRIC80

RECOGNITION ROBOTICS
The Visual Guidance Company

C++ Day 2017
2 Dicembre, Modena

Anyone not using smart ptrs?

Smart Pointers are cool

- (good) smart pointers enable **explicit lifetime design**
- Still smart pointers have some run-time overhead
- Talk is focusing on that overhead, not lifetime design
- Specifically, shared pointers

Just a proof of concept

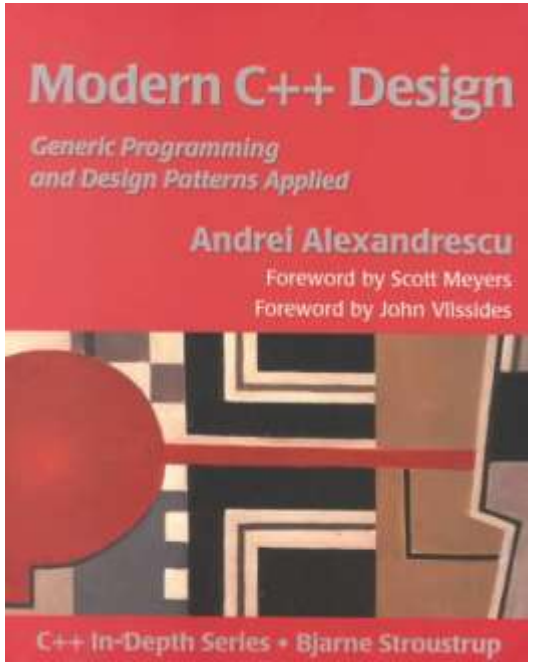
- Code lacks support for polymorphic conversion etc.
- I’ve kept this thing in a closet way too long :-)
- Probably the “invention” process is more interesting than the thing itself.

Shared pointers

- Shared ownership of an object
- Last pointer to go destroys the pointed object
- Usually keeps a count of incoming references to obj
- Count must be shared as well

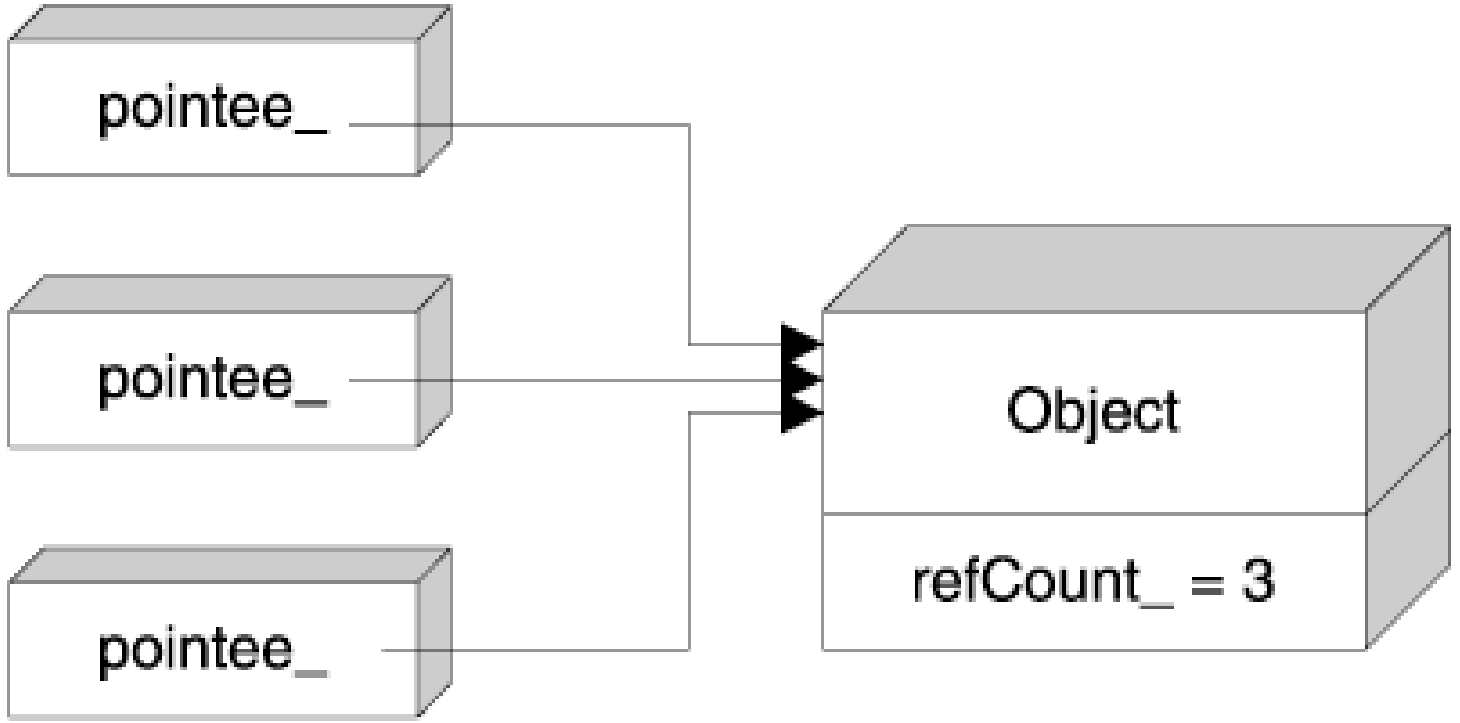
Shared pointers in practice

- 3 basic implementations
- (+ an uncommon one)

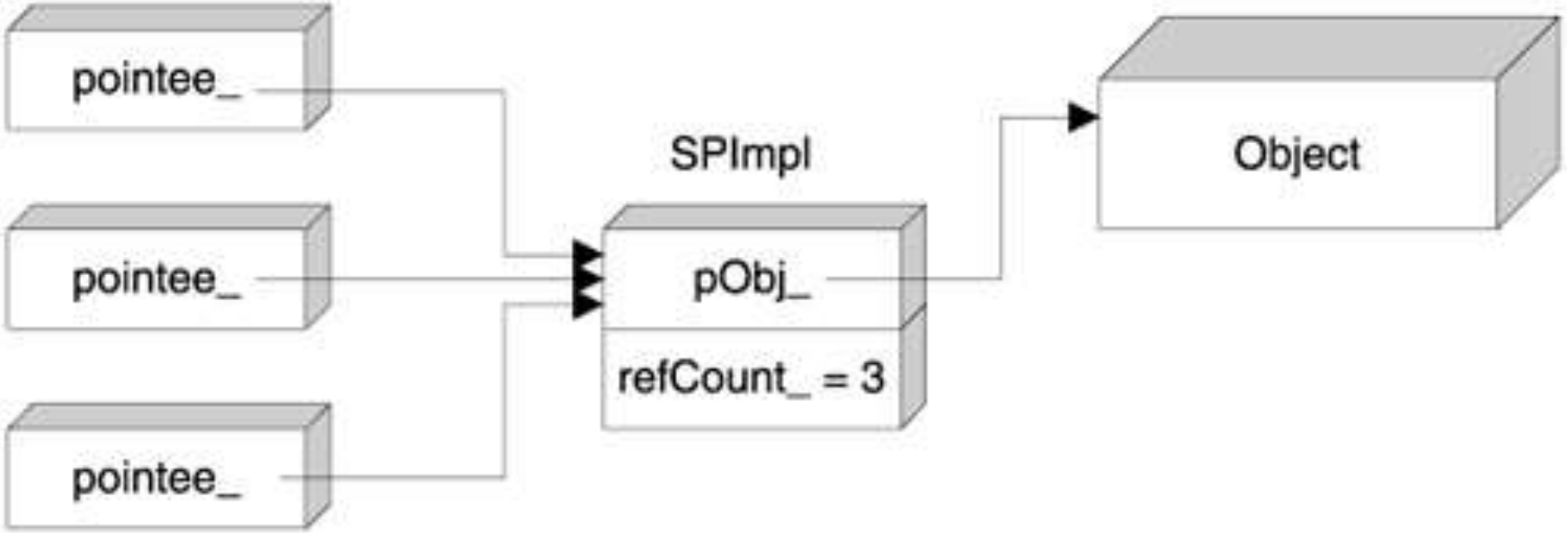


- C++11 implementation

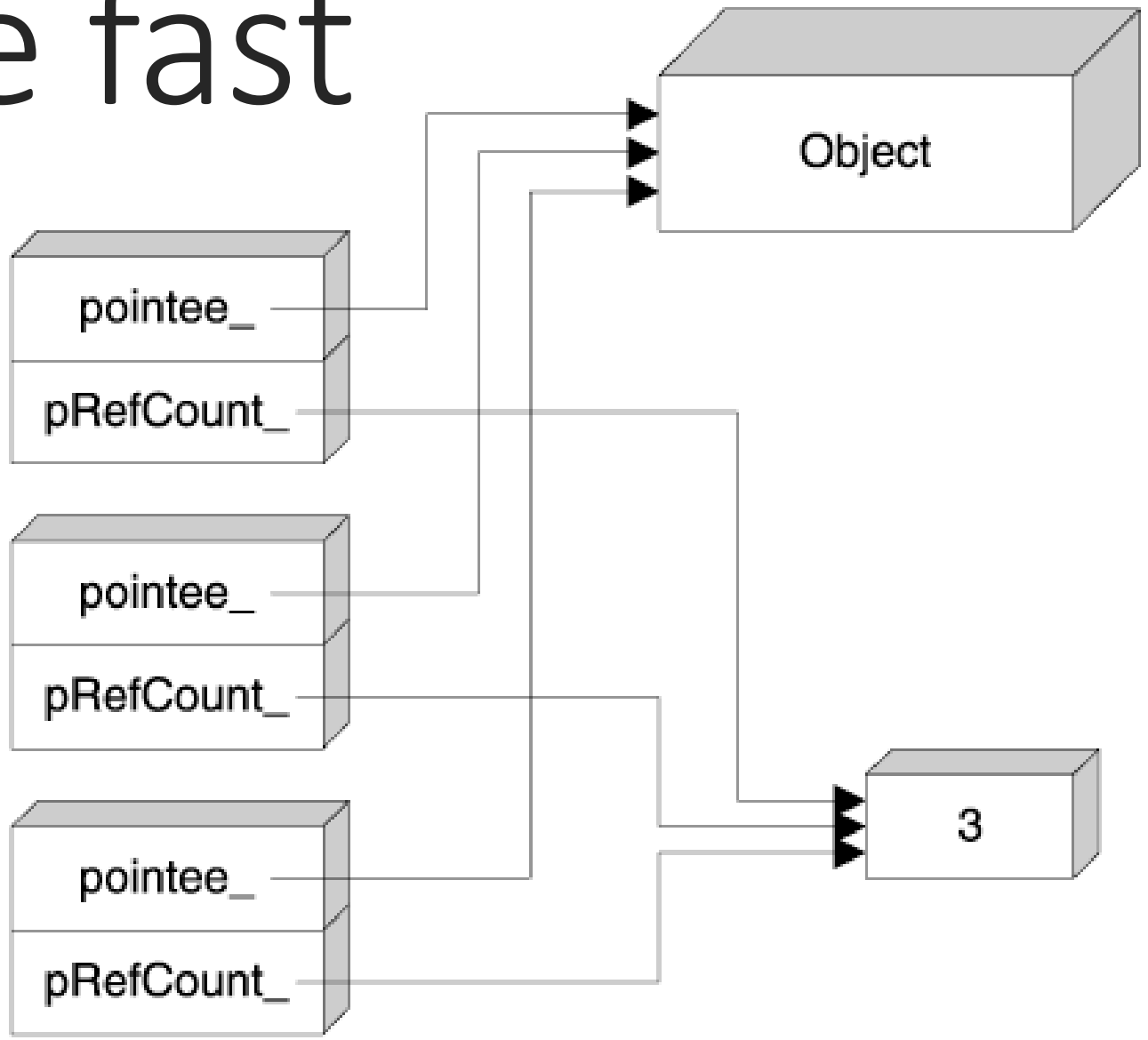
Invasive



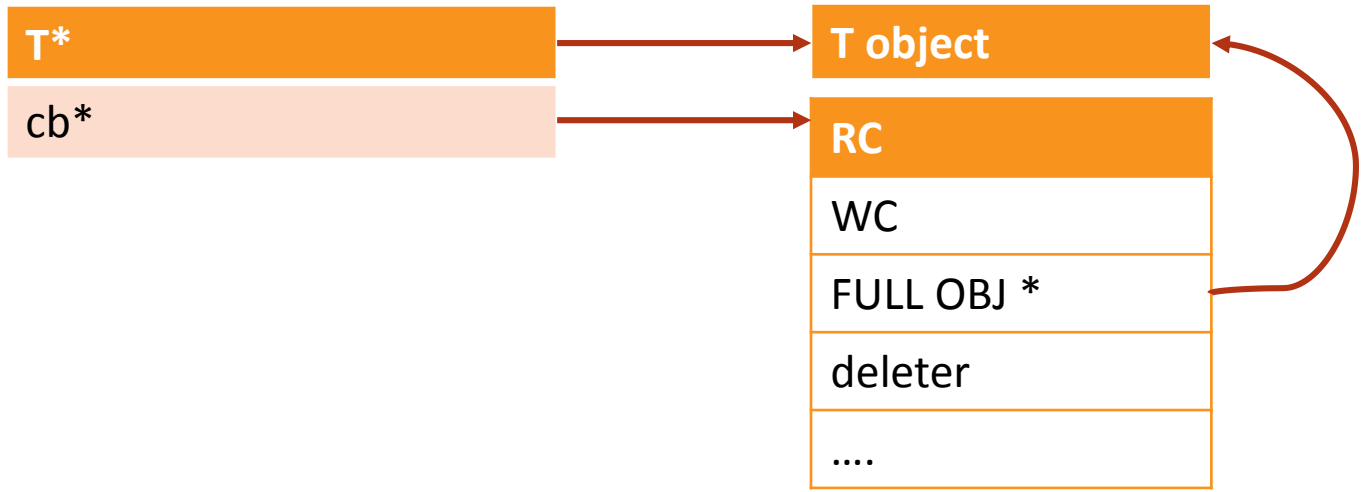
Slim, maybe slow



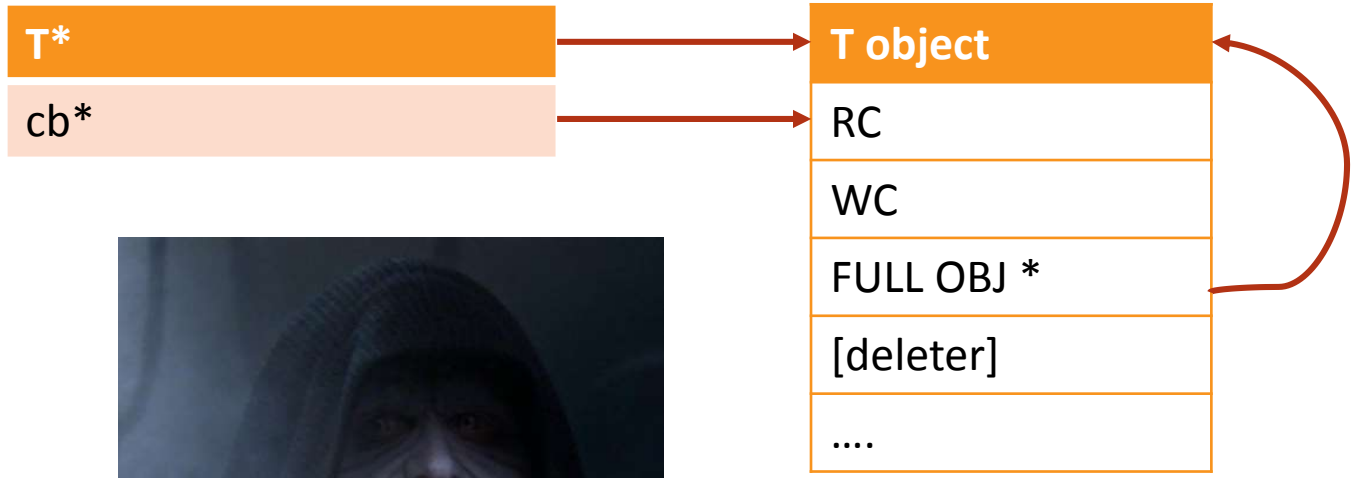
Fat, maybe fast



Standard



Standard + make_shared



+

<Interlude>

Rant is coming

Why I don’t like shared_ptr

OPINION <<< BEWARE :)

“C was designed on the assumption that the programmer is someone sensible who knows what he's doing”

(Kernighan and Ritchie)

“What you don't use, you don't pay for” (Stroustrup)

shared_ptr: the dark side

You pay the overhead of weak even if you don't use it

Trying to protect you from absence of virtual destructor

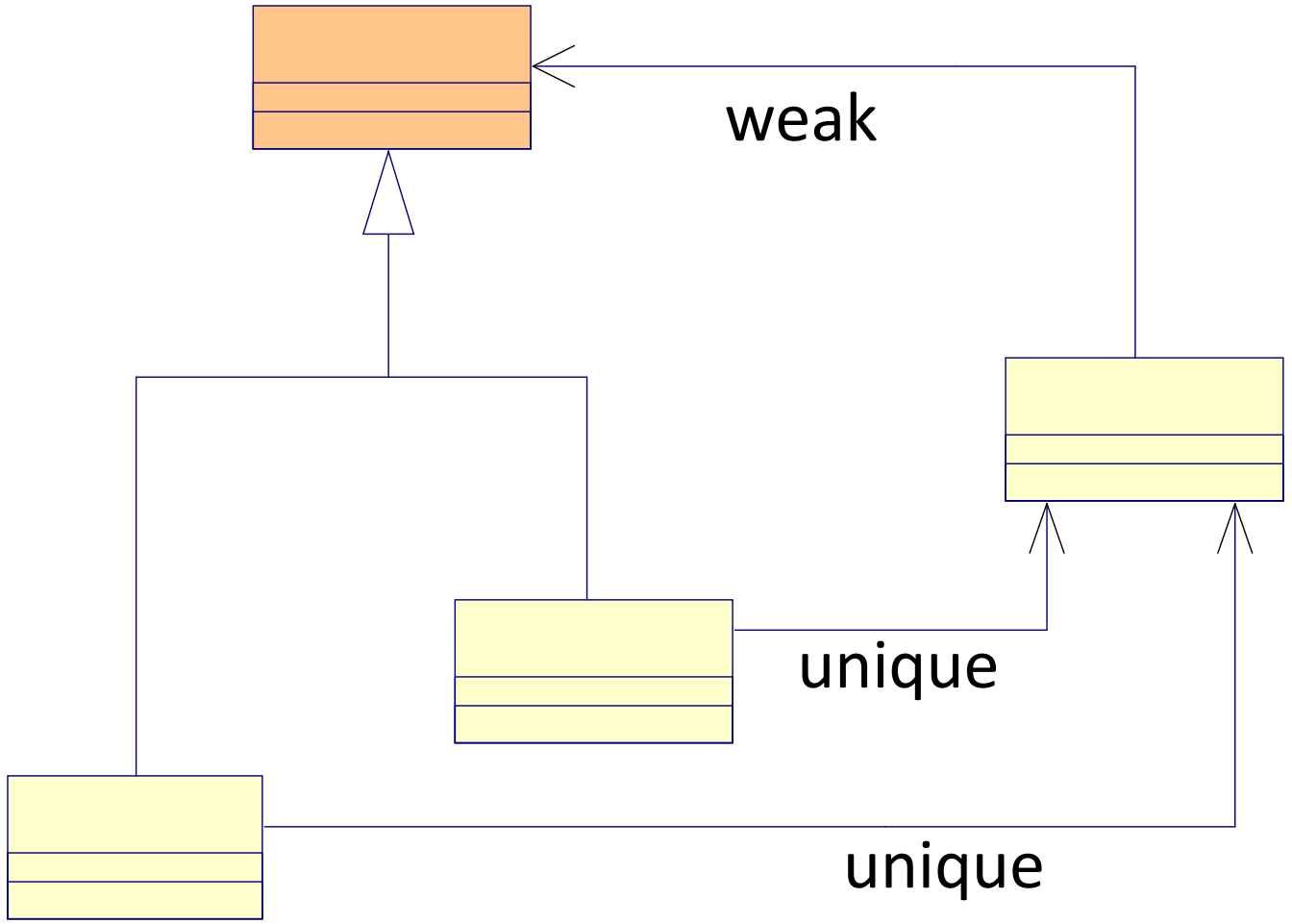
You lose a standalone weak ptr

make_shared is a virus (factory -> ~~unique~~ -> penalty)

Safe-by-design weak usage

NOT VALID C++

(just an example)



</Interlude>

Sorry for ranting

A little benchmark

- Not much science, just a few reasonable tests, should be expanded, tested with more compilers, etc.
- 3 tests: allocation, dereferencing, “complex” usage.
- Tested raw, std::shared_ptr, std:shared_ptr + make_shared, a slim and a fat version I implemented.

Baseline (raw pointers)

```
struct Data
{
public:
    Data(int v) : n(v)
    {
        ++liveInstances;
    }

    ~Data()
    {
        --liveInstances;
    }

    // …

private:
    static int liveInstances;
    int n;
};
```

```
struct Lt
{
    bool operator()(const Data* d1, const Data* d2)
    {
        return d1->Value() < d2->Value();
    }
};
```

Test 1

```
vector<Data*> v0;
v0.reserve(N);
for (int j = 0; j < 10; ++j)
{
    for (int i = 0; i < N; ++i)
        v0.push_back(new Data(i));
    for (int i = 0; i < N; ++i)
        delete v0[i];
    v0.clear();
}
```

N = 500000

Just a lot of construction / destruction, no usage

Test 2

V1 prefilled with N random values.
n used after to prevent optimization
Lot of usage, no creation / destructions

```
for (int j = 0; j < 1000; ++j)
    {
        for (int i = 0; i < N; ++i)
            n += v1[i]->Value();
    }
```

Test 3

(playing around)

```
reverse(v1.begin(), v1.end());
random_shuffle(v1.begin(), v1.end());
sort(v1.begin(), v1.end(), Lt());
int m =
(*(max_element(v1.begin(), v1.end(), Lt())))->Value();

sort(v2.begin(), v2.end(), Lt());

auto bi = back_inserter(v3); // reserved at N not 2N
merge(v1.begin(), v1.end(),
      v2.begin(), v2.end(), bi, Lt());

v4.assign(v3.begin(), v3.end());  // reserved at 2N
```

Notes:

- Care is needed with delete: v1 and v2 need a delete cycle, v3 and v4 do not because they keep copies of raw ptrs.

- When using smart ptrs:

```
struct Lt
{
    // intentionally without a & to trigger more copies!
    bool operator()(const shared_ptr<Data> r1, const shared_ptr<Data> r2)
    {
        return r1->Value() < r2->Value();
    }
};
```

Results

ms

MB

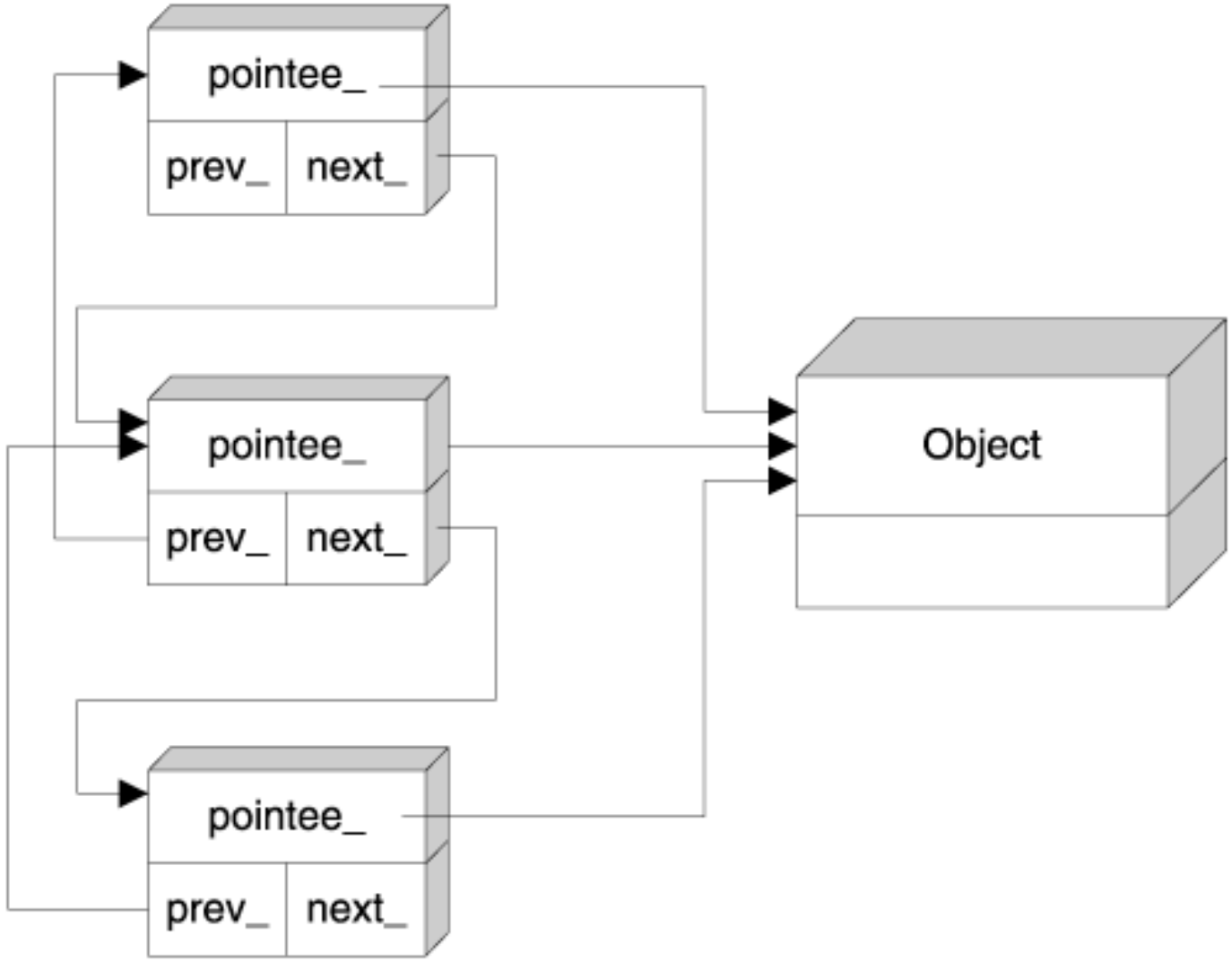
	test 1	test 2	test 3	max RAM (VS)
raw	362	881	238	109
fat	682	1614	643	217
slim	793	2481	859	200
std	835	1099	1584	253
make_shared	505	1795	1459	181

Can we make it faster?

- The weird one
- Some physics of software (run-time space)
- Use the physics, Luke

“Reference Linking”

2001



Look, no counter…

Constructor won't throw

as it does not allocate extra storage

No need for make_shared

no need to use shared ptrs everywhere

Fat (200% overhead)

Scale-free Geometry in Object-Oriented Programs

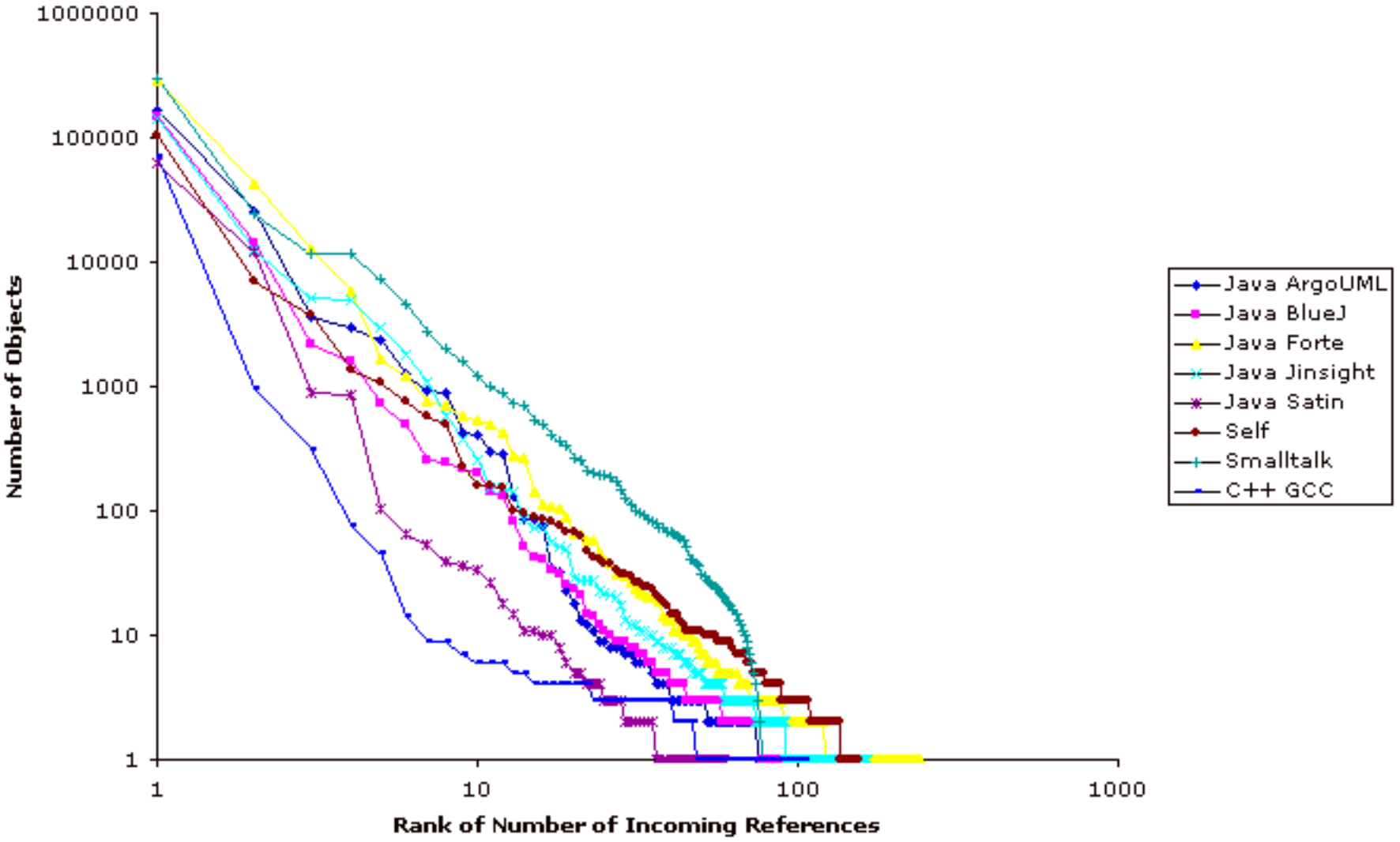
Alex Potanin, James Noble, Marcus Frean, Robert Biddle
School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
{alex, kjx, marcus, robert}@mcs.vuw.ac.nz

2005

Introduction

Object-oriented programs, when executed, produce a complex web of objects that can be thought of as a graph with objects as nodes and references as edges. In recent years interest has grown in the geometry of networks (or graphs), particularly those of human origin, many of which show a rather striking property: their structure is *scale-free*. In the case of the World Wide Web, for example, the number of web pages with 1 incoming link is about twice the number with 2 incoming links, and *that* is twice the number with 4 links, and so on all the way

Incoming count ranks



Quick check

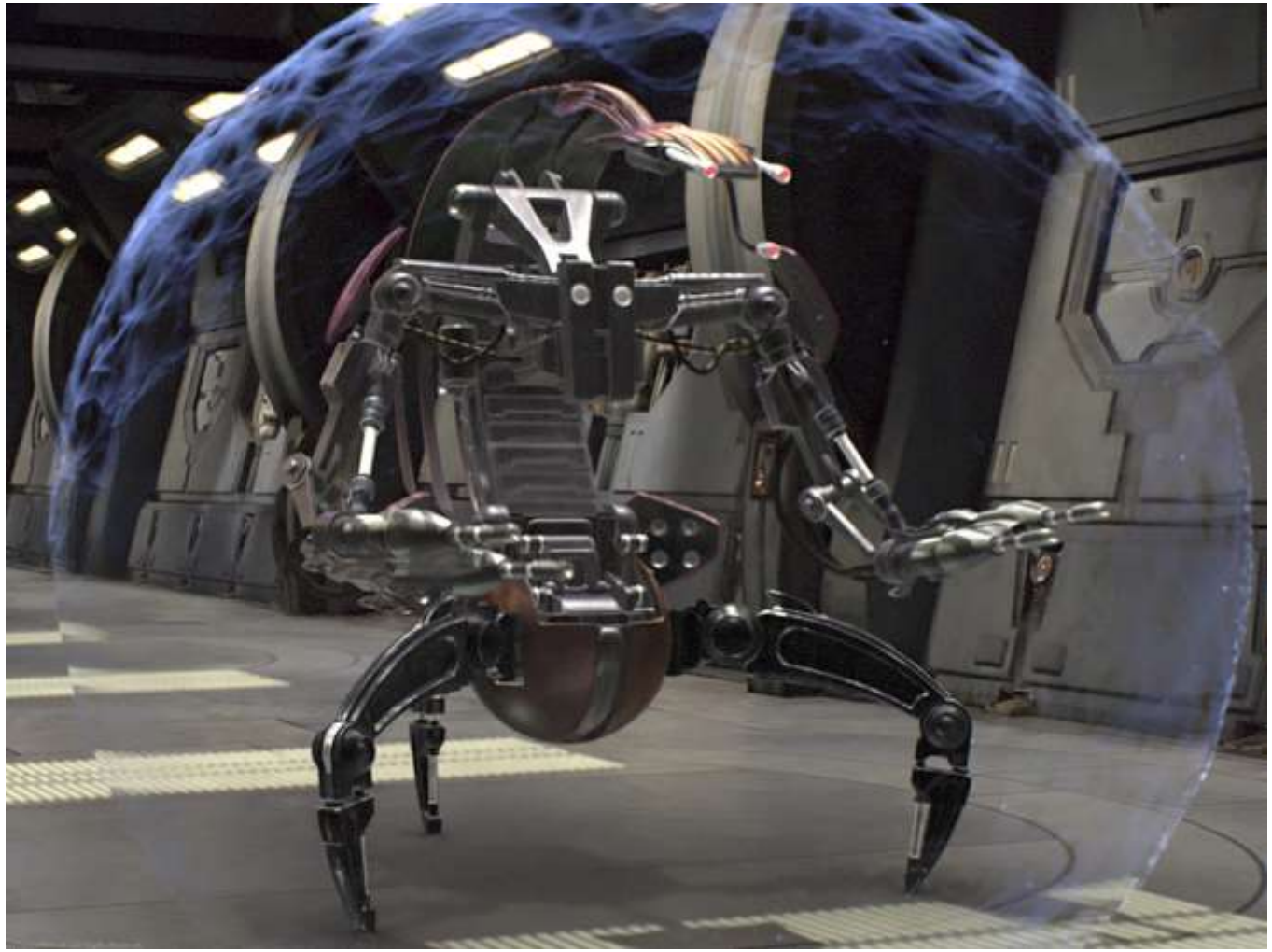
Max counter was **3**

in the previous

benchmark

(tested in my FAT

implementation)



So?

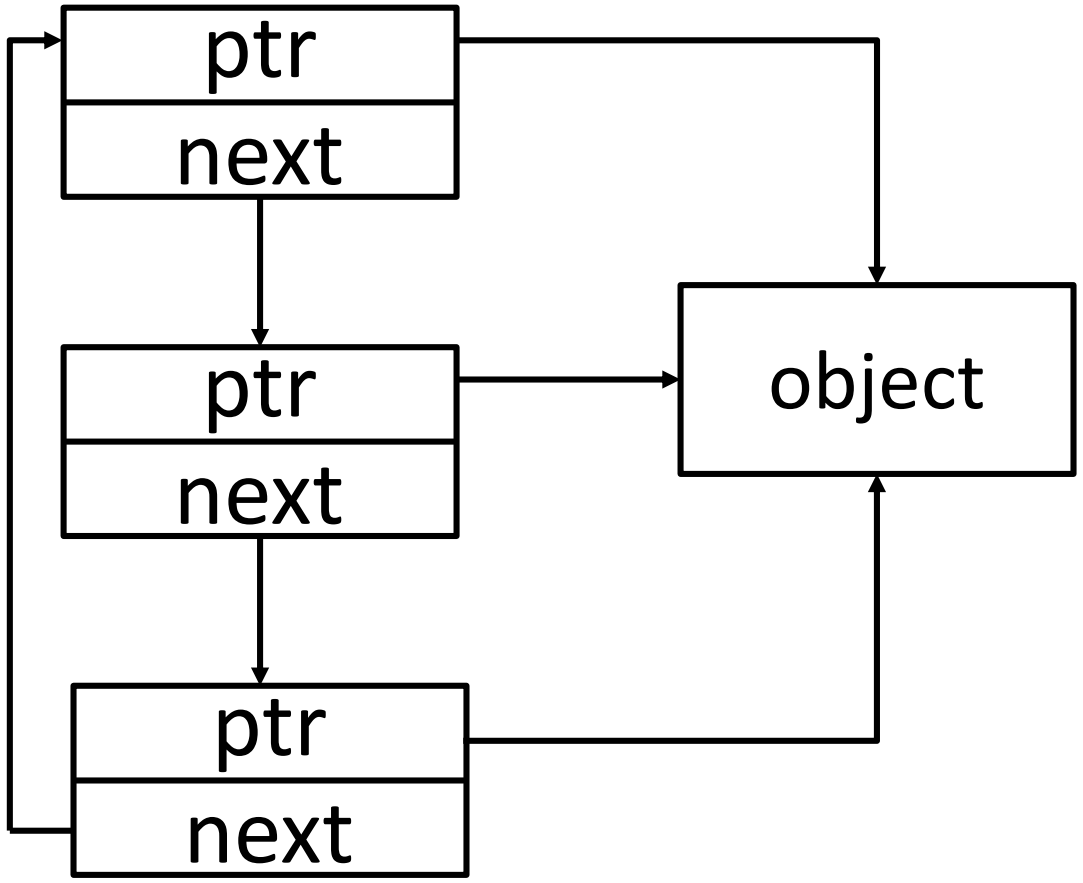
Two steps:

1) Optimize for the small

2) Turn into FAT when count > K, K small (3-4)

Must keep the same memory footprint between 1 -> 2

1) Optimize for the small



Trivial implementation

Single linked list

Predecessor -> while loop (short path anyway)

Alternatives / experiments needed

watch code size / inlining

The public stuff (some)

```
template< class T > class LinkedSmallPtr
{
public:
    LinkedSmallPtr()
    {
        p = nullptr;
        next = this;
    }

    explicit LinkedSmallPtr(T* t) noexcept
    {
        p = t;
        next = this;
    }

    LinkedSmallPtr(const LinkedSmallPtr& rhs)
    {
        Copy(rhs);
    }

    LinkedSmallPtr(LinkedSmallPtr&& rhs)
    {
        MoveFrom(rhs);
    }

    ~LinkedSmallPtr()
    {
        RemoveLink();
    }
```

The public stuff (some)

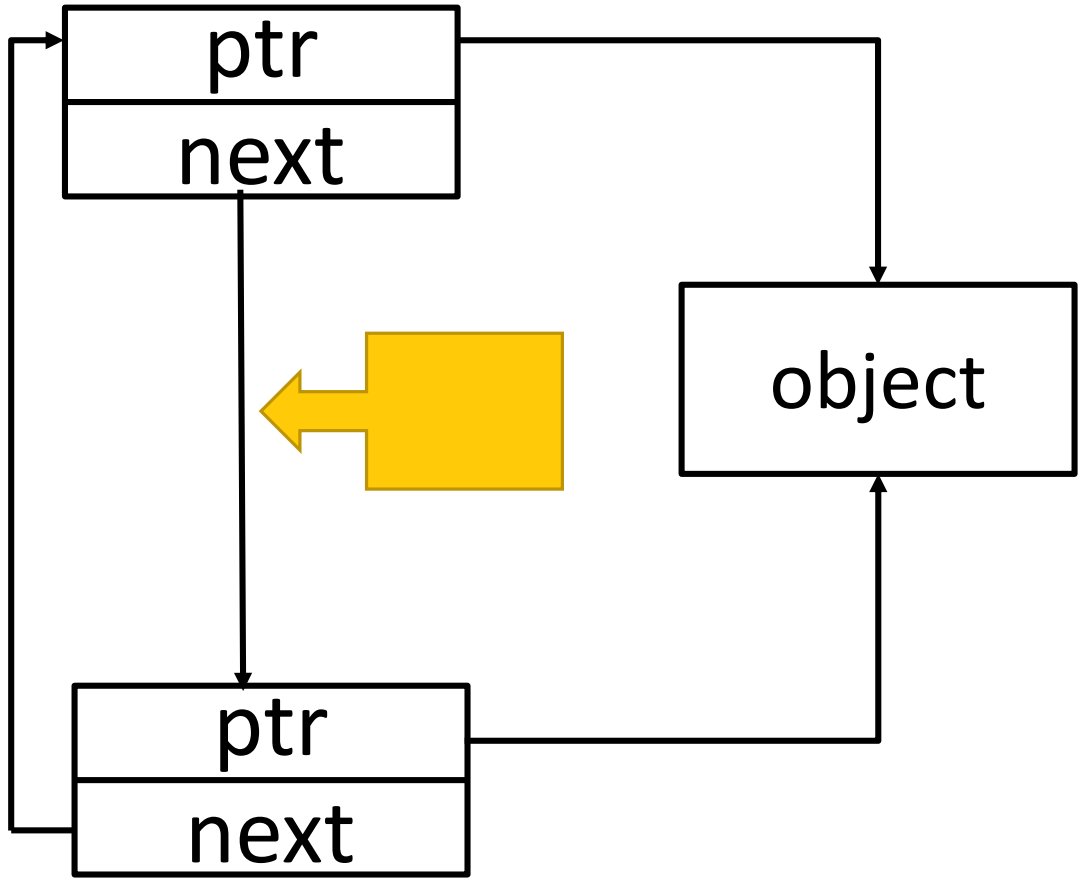
```
LinkedSmallPtr& operator =(const LinkedSmallPtr& rhs)
    {
        if (&rhs != this)
        {
            RemoveLink();
            Copy(rhs);
        }
        return *this;
    }

LinkedSmallPtr& operator=(LinkedSmallPtr&& rhs)
    {
        if (&rhs != this)
        {
            RemoveLink();
            MoveFrom(rhs);
        }
        return *this;
    }
```

The private stuff

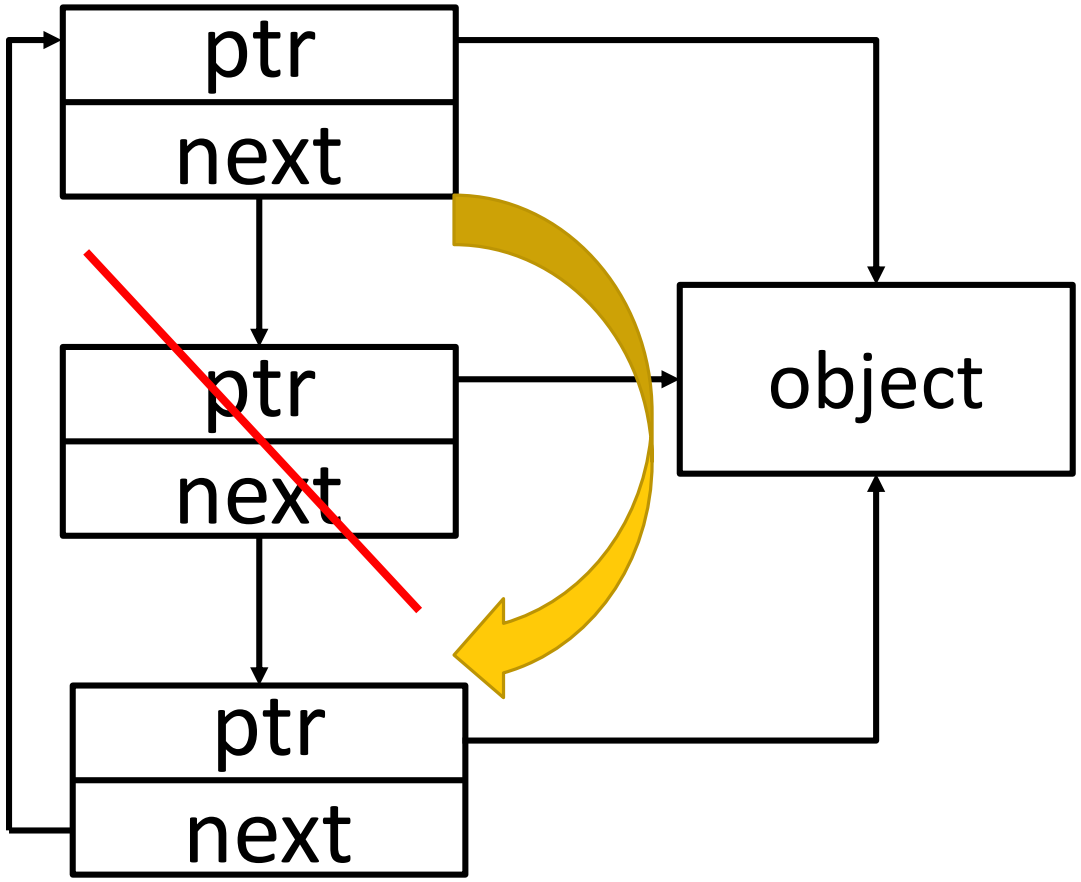
```
private:
     T* p;
     mutable LinkedSmallPtr* next;

     void Copy(const LinkedSmallPtr& rhs)
     {
          p = rhs.p;
          next = rhs.next;
          rhs.next = this;
     }
```



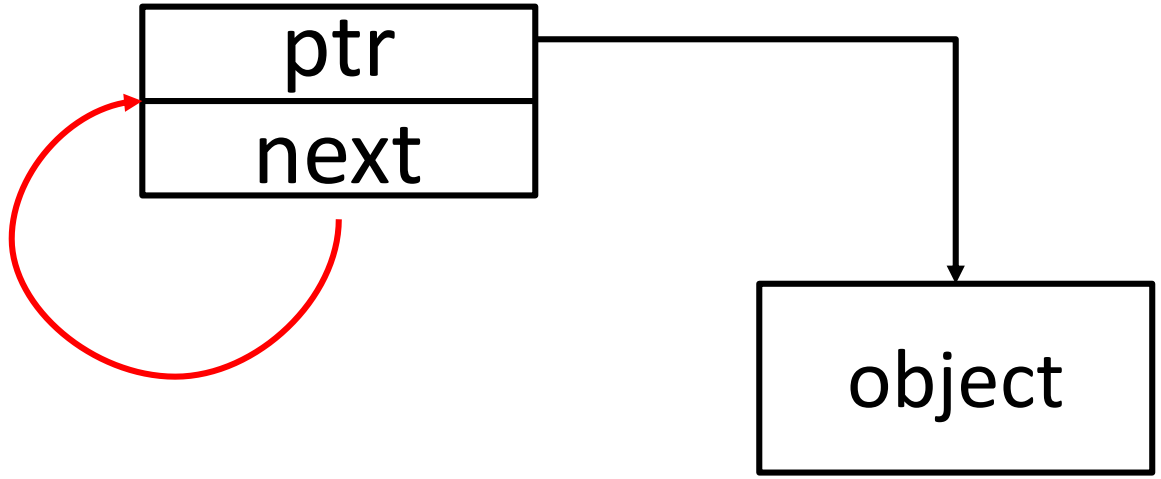
The private stuff

```
void RemoveLink()
    {
        if (next == this)
        {
            delete p;
        }
        else
        {
            LinkedSmallPtr* h = next;
            while (h->next != this)
                h = h->next;
            h->next = next;
        }
    }
```



The private stuff

```
void MoveFrom(LinkedSmallPtr& rhs)
    {
        p = rhs.p;
        if (rhs.next == &rhs)
        {
            next = this;
        }
        else
        {
            next = rhs.next;
            LinkedSmallPtr* h = next;
            while (h->next != &rhs)
                h = h->next;
            h->next = this;
        }
        rhs.p = nullptr;
    }
```



Benchmark

ms MB

	test 1	test 2	test 3	max RAM (VS)
raw	362	881	238	109
fat	682	1614	643	217
slim	793	2481	859	200
std	835	1099	1584	253
make_shared	505	1795	1459	181
small linked	339	1089	633	146

2) Highly-Adaptive Scavenger

It’s “linked” when count is < K

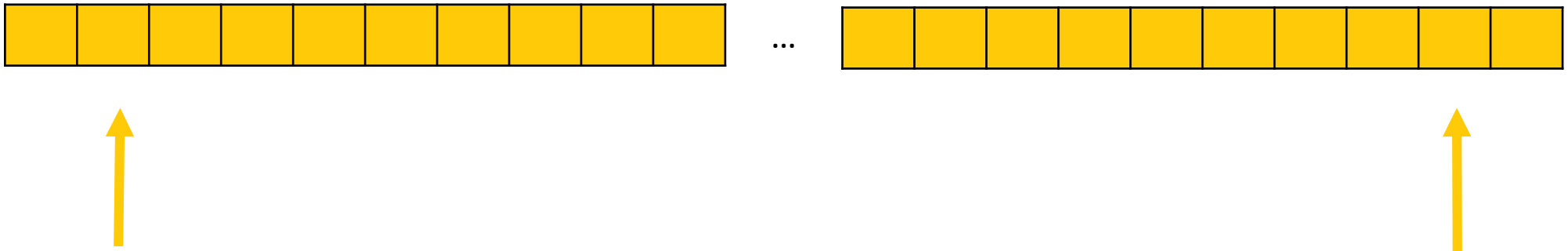
Becomes ref counted after K

While keeping its storage / identity

Uses bits you’re normally wasting

Wasted bits

and where to find them



Excessive address space (e.g. PIC32 w/ a few MB of memory)

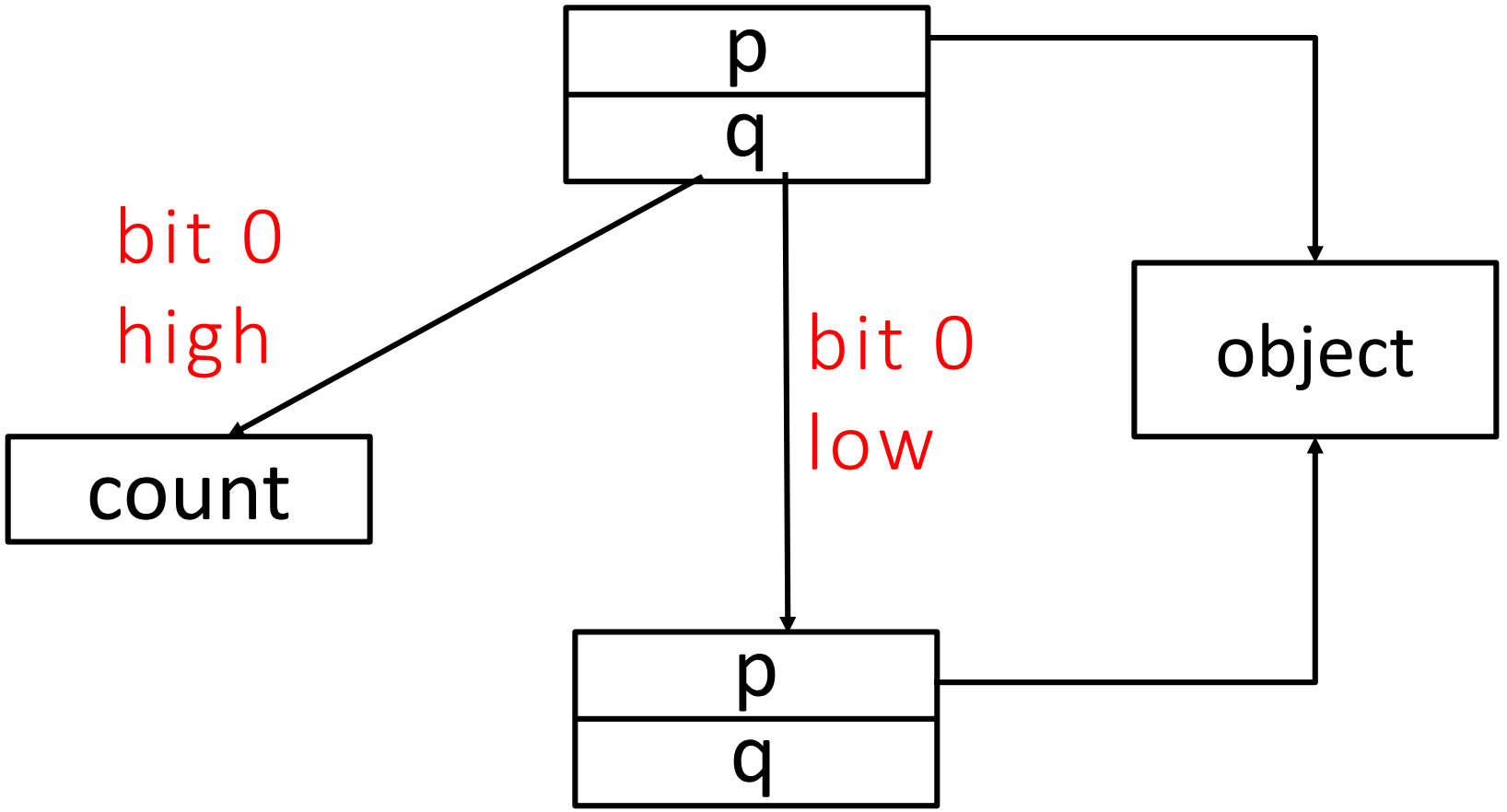
Alignment (pointers to int, long, float, double, …)

Alignment (useless :-)

```
struct alignas(2) alignas(sizeof(void*)) RefCount
{
   int count;
};

template< class T >
class alignas(2) alignas(sizeof(void*)) LinkedSmallPtr
{
   // …
```

Basic strategy



Ugly, step 1

```
private:
    T* p;
    mutable uintptr_t next;
```

can’t play with bits on pointers…

How do you “count”?

On copy: hard / probably inefficient / more bits (?)

On destruction: easy / statistically ok

Better ideas welcome ☺

Ugly, step 2

```
void RemoveLink()  {
        if ((next & 1) == 0)  {
            LinkedSmallPtr* lNext = reinterpret_cast<LinkedSmallPtr*>(next);
            if (lNext == this) {
                delete p;
            }
            else  {
                int count = 0;
                LinkedSmallPtr* h = lNext;
                LinkedSmallPtr* hn;
                while ((hn=reinterpret_cast<LinkedSmallPtr*>(h->next)) != this)  {
                    h = hn;
                    ++count;
                }
                h->next = next;
                if (count > THRESHOLD)
                    assert(false);
            }
        }
        else
            assert(false);
    }
```

FIX OTHERS, not ME ☺

“Promotion cost”

Once you get over threshold, you need to remap ALL the nodes from linked to counted.

Careful not to oscillate around this point!

If you grow big, you stay big

- Once you get promoted from linked to counted, you stay counted even if the count goes down.

You become your rhs

CC	RHS	
	L	C
	L	C

CA	RHS	
	L	C
LHS L	L	C
LHS C	L	C

MC	RHS	
	L	C
	L	C

MA	RHS	
	L	C
LHS L	L	C
LHS C	L	C

~	L	C
	L	C

Benchmark

ms

	test 1	test 2	test 3
raw	362	881	238
small linked	339	1089	633
adaptive linked	345	1132	639

What's the catch?

Code is significantly more complex

simplification welcome

Degenerate behavior potentially costly

reference a lot, then release all

Conclusions

The “big” idea is building something that responds well to the most common real-world scenarios, while adapting to the “long tail” of unusual cases.

Extends well beyond this case

lots of experiments still needed

Maturity scale

- Everything is an array / a list
- Choose the right data structure / algo
  - fake: map vs roll your own
  - real: convenient structures for small / large cases

Frequent / Rare

- Adaptive data structures with stable ifc <<<
- Self-tuning data structures with stable ifc

Keep exploring :-)

Brain-Driven Design

carlo.pescio@gmail.com