

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this correct?  
void f()  
{  
    char *cp = new char[1024];  
  
    g(cp);  
    h(cp);  
  
    delete [] cp;  
}
```

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this maintainable?  
void f()  
{  
    char *cp = new char[1024];  
  
    try  
    {  
        g(cp);  
        h(cp);  
        delete [] cp;  
    }  
    catch (...)  
    {  
        delete [] cp;  
        throw;  
    }  
}
```

RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
};

void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be freed here
```

RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
}; // Copy? Move?

void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be freed here
```

RAII

- Should be careful when implementing RAII
- Destructor calls only when **living object** goes out of scope
- Object lives only when constructor has successfully finished

// But be careful:

```
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
        /* ... */ if (error) throw XXX; /* ... */
    }
};
```

Smart pointers

- The deprecated auto pointer
- How smart pointers work?
- `Unique_ptr`
- `Shared_ptr` and `weak_ptr`
- `Make_` functions
- Shared pointer from this
- Traps and pitfalls

How to handle ownership?

- Owner
 - Responsible releasing the resource
 - Single owner
 - Reference counting
 - Examples
 - Memory `std::vector<>`, `std::array<>`, `std::shared_ptr<>`, `std::string`
 - `std::lock_guard<>`
 - `std::ifstream`
- Observer
 - Examples
 - `std::weak_ptr<>`
 - `std::string_view<>`

Auto_ptr

- The only smart pointer in C++98/03
- Cheap, ownership-based
- Not works well with STL containers and algorithms
- Not works with arrays
- Deprecated in C++11
- **Removed from C++ since C++17**

Unique_ptr

- Single ownership pointer (similar to auto_ptr)
- Carefully designed to work with STL and other language features
- Movable but not copyable
- Deleter is type parameter – cannot be changed in run-time

```
#include <memory>
template< class T, class Deleter = std::default_delete<T>>
class unique_ptr;

template <class T, class Deleter>
class unique_ptr<T[],Deleter>;

void f()
{
    std::unique_ptr<MyClass>    up1(new MyClass()); // * and ->
    std::unique_ptr<MyClass[]> up2(new MyClass[n]); // []
    ...
} // proper delete called here
```

Unique_ptr

```
#include <memory>
```

```
void f()
```

```
{
```

```
    std::unique_ptr<Foo> up(new Foo{}); // up is the only owner  
    std::unique_ptr<Foo> up2(up); // compile error: can't copy unique_ptr  
    std::unique_ptr<Foo> up3; // nullptr: not owner  
    const std::unique_ptr<Foo> up4(new Foo{}); // const pointer
```

```
    up3 = up; // compile error: can't assign unique_ptr  
    up3 = std::move(up); // ownership moved to up3  
    up3 = std::move(up4); // compile error: const can't move
```

```
    std::vector<unique_ptr<int>> vi;  
    vi.push_back(std::make_unique<int>(42));
```

```
} // vi destroyed: int{42} is destroyed  
    // up4 destroyed: Foo object is destroyed  
    // up3 destroyed: Foo object is destroyed  
    // up destroyed: nop
```

How inheritance is implemented?

- Raw pointers: assign Derived* to Base* works
- But unique_ptr<Derived> is not inherited from unique_ptr<Base>
- The deleter is not copied – different from shared_ptr !!!

```
template<class T, class D>
class unique_ptr
{
private:
    T* ap;    // refers to the actual owned object (if any)
public:
    typedef T element_type;

    explicit unique_ptr (T* ptr = 0) : ap(ptr) { }
    unique_ptr (unique_ptr&& rhs) : ap(rhs.release()) { }
    template<class Y> unique_ptr(unique_ptr<Y>&& rhs):ap(rhs.release()){}

    unique_ptr& operator=(unique_ptr&& rhs) { ... }
    template<class Y, class D>
    unique_ptr& operator=(unique_ptr<Y,D>&& rhs) { ... }
};
```

Polymorphism

```
struct Base {
    virtual void f() { std::cout << "Base::f\n"; }
    ~Base() { std::cout << "Base::~~Base()\n"; }
};
struct Derived : Base {
    virtual void f() override { std::cout << "Derived::f\n"; }
    ~Derived() { std::cout << "Derived::~~Derived()\n"; }
};

void g() {
    auto dp = std::make_unique<Derived>(); // std::unique_ptr<Derived>
    std::unique_ptr<Base> bp = std::move(dp);

    std::vector<std::unique_ptr<Base>> v;
    v.push_back(std::make_unique<Base>());
    v.push_back(std::make_unique<Derived>());
    for ( auto& p : v ) p->f();
};
```

Polymorphism

```
struct Base {
    virtual void f() { std::cout << "Base::f\n"; }
    ~Base() { std::cout << "Base::~~Base()\n"; }
};
struct Derived : Base {
    virtual void f() override { std::cout << "Derived::f\n"; }
    ~Derived() { std::cout << "Derived::~~Derived()\n"; }
};
```

```
void g() {
    auto dp = std::make_unique<Derived>();
    std::unique_ptr<Base> bp = std::move(dp);

    std::vector<std::unique_ptr<Base>> v;
    v.push_back(std::make_unique<Base>());
    v.push_back(std::make_unique<Derived>());
    for ( auto& p : v ) p->f();
};
```

```
Base::f           // v[1] is Base
Derived::f       // v[2] is Derived
Base::~~Base()   // oops v[2] is Derived during destruction of v
Base::~~Base()   // ok v[1] is Base during destruction of v
Base::~~Base()   // oops delete bp
```

Polymorphism

```
struct Base {
    virtual void f() { std::cout << "Base::f\n"; }
    virtual ~Base() { std::cout << "Base::~~Base()\n"; }
};
struct Derived : Base {
    virtual void f() override { std::cout << "Derived::f\n"; }
    virtual ~Derived() override { std::cout << "Derived::~~Derived()\n"; }
};
```

```
void g() {
    auto dp = std::make_unique<Derived>();
    std::unique_ptr<Base> bp = std::move(dp); // deleter is not copied

    std::vector<std::unique_ptr<Base>> v;
    v.push_back(std::make_unique<Base>());
    v.push_back(std::make_unique<Derived>()); // deleter is not copied
    for ( auto& p : v ) p->f();
};
```

```
Base::f // v[1] is Base
Derived::f // v[2] is Derived
Base::~~Base() // base part of v[2] Derived
Derived::~~Derived() // derived part of v[2] since virtual destructor
Base::~~Base() // v[1]
Derived::~~Derived() // bp points to Derived with virtual destructor
Base::~~Base() // base part of *bp
```

Polymorphism – shared_ptr

```
struct Base {
    virtual void f() { std::cout << "Base::f\n"; }
    virtual ~Base() { std::cout << "Base::~~Base()\n"; }
};
struct Derived : Base {
    virtual void f() override { std::cout << "Derived::f\n"; }
    virtual ~Derived() override { std::cout << "Derived::~~Derived()\n"; }
};

void g() {
    auto dp = std::make_shared<Derived>();
    std::shared_ptr<Base> bp = std::move(dp); // deleter is moved

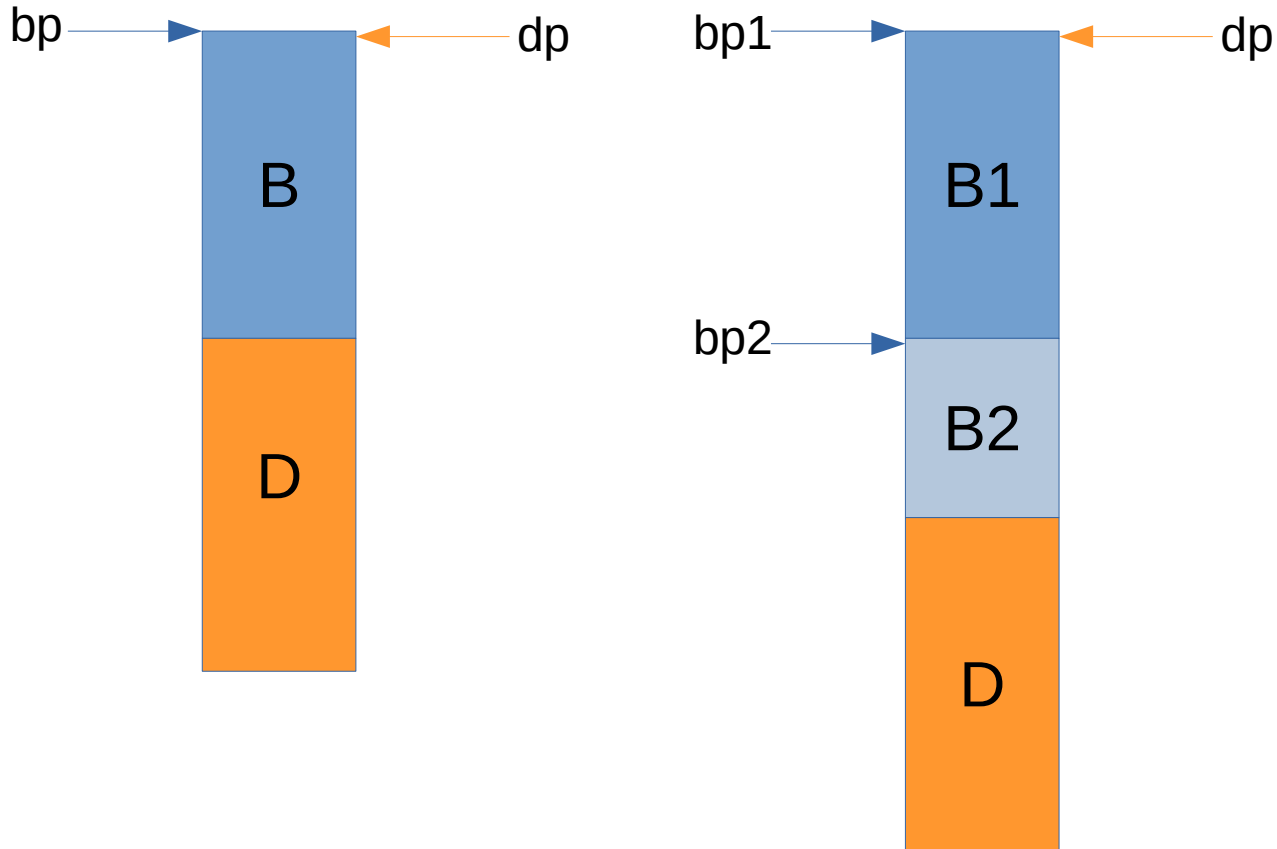
    std::vector<std::shared_ptr<Base>> v;
    v.push_back(std::make_shared<Base>());
    v.push_back(std::make_shared<Derived>()); // deleter is copied
    for ( auto& p : v ) p->f();
};

Base::f // v[1] is Base
Derived::f // v[2] is Derived
Base::~~Base() // base part of v[2] Derived
Derived::~~Derived() // derived part of v[2] since default_deleter<Derived>
Base::~~Base() // v[1]
Derived::~~Derived() // bp has default_deleter<Derived>
Base::~~Base() // base part of *bp:
```

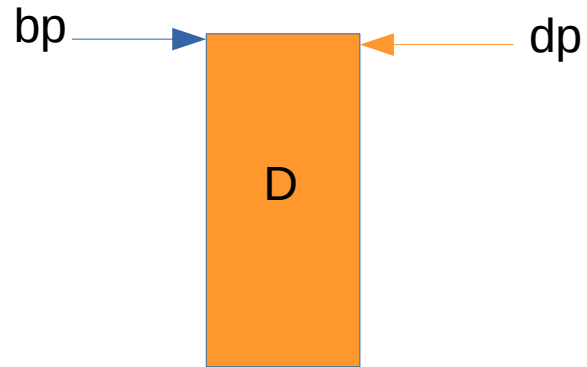
Default deleter empty base optimization

```
template<class _T, class _DeleterT = std::default_delete<_T>>
class unique_ptr
{
public:
    // public interface...
private:
    // using empty base class optimization to save space
    // making unique_ptr with default_delete the same size as pointer
    class _UniquePtrImpl : private _DeleterT
    {
public:
        constexpr _UniquePtrImpl() noexcept = default;
        // some other constructors...
        deleter_type& _Deleter() noexcept { return *this; }
        const deleter_type& _Deleter() const noexcept { return *this; }
        pointer& _Ptr() noexcept { return _MyPtr; }
        const pointer _Ptr() const noexcept { return _MyPtr; }
private:
        pointer _MyPtr;
    };
    _UniquePtrImpl _MyImpl;
};
```

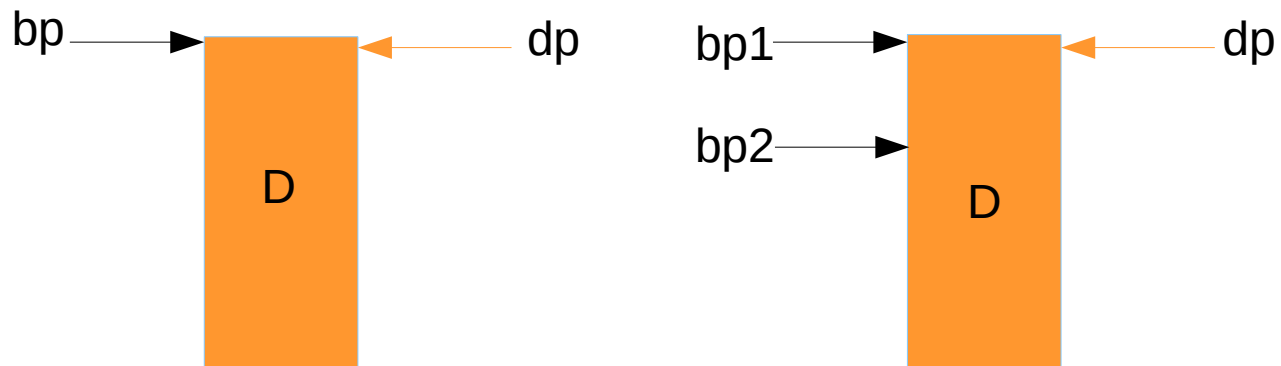
Structure of (sub)objects



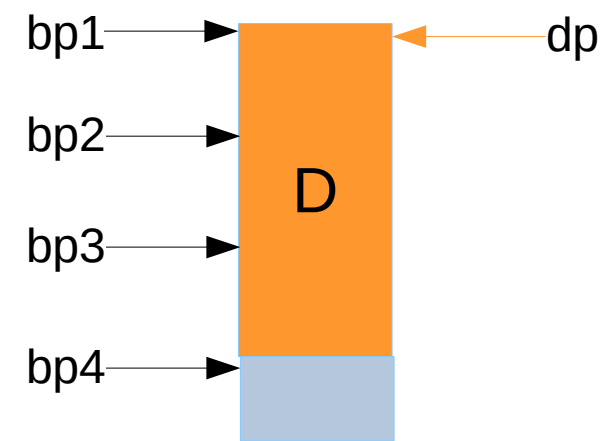
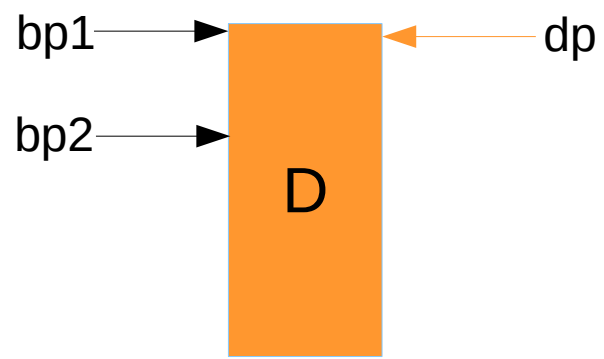
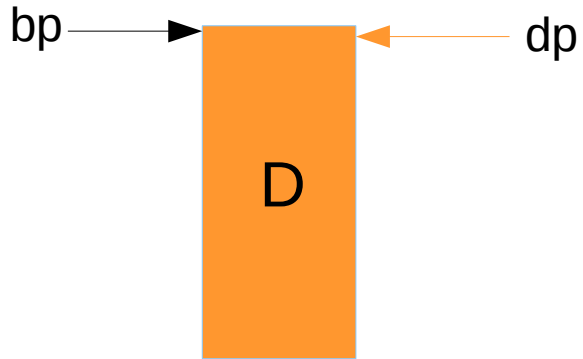
Empty base



Empty base



Empty base



Abstract factory pattern

```
#include <memory>

class Base { ... };
class Derived1 : public Base { ... };
class Derived2 : public Base { ... };

template <typename... Ts>
std::unique_ptr<Base> makeBase( Ts&&... params ) { ... }

void f() // client code:
{
    auto pBase = makeBase( /* arguments */ );
}
// destroy object
```

Abstract Factory Pattern

```
auto delBase = [](Base *pBase)
{
    makeLogEntry(pBase);
    delete pBase; // delete object
};

template <typename... Ts>
std::unique_ptr<Base, decltype(delBase)> makeBase( Ts&&... params)
{
    std::unique_ptr<Base, decltype(delBase)> pBase(nullptr, delBase);

    if ( /* Derived1 */ )
    {
        pBase.reset(new Derived1( std::forward<Ts>(params)... ) );
    }
    else if ( /* Derived2 */ )
    {
        pBase.reset(new Derived2( std::forward<Ts>(params)... ) );
    }
    return pBase;
}
```

Evaluation

- Can be used in standard containers when polymorphic use needed
- The `sizeof(unique_ptr<>)` is `sizeof(raw pointer) + deleter size`
- If `default_deleter` is used, then no extra size penalty
- If no deleter state (e.g. lambda with no capture): `+ sizeof(funptr)`
- If deleter with state, the size increases
- Prefer `unique_ptr` when possible
- No copy of deleter :(
- No downcast operation :(

Downcast unique_ptr

```
template<typename Derived, typename Base, typename Del>
std::unique_ptr<Derived, Del>
dynamic_unique_ptr_cast( std::unique_ptr<Base, Del>&& p )
{
    if(Derived *r = dynamic_cast<Derived *>(p.get()))
    {
        p.release();
        return std::unique_ptr<Derived, Del>(r, std::move(p.get_deleter()));
    }
    return std::unique_ptr<Derived, Del>(nullptr, p.get_deleter());
}
```

shared_ptr

- Shared ownership pointer with reference counter
- Copy constructible and assignable
- Array specializations (`shared_ptr<T[]>`) since C++17
- Deleter type parameter – “copied”

```
#include <memory>
template< class T, class Deleter = std::default_delete<T>>
class shared_ptr;

void f()
{
    std::shared_ptr<MyClass>    sp1(new MyClass()); // * and ->
    std::shared_ptr<MyClass>    sp2(new MyClass[n], // * and ->
                                   std::default_delete<MyClass[ ]>()); // before C++17
    std::shared_ptr<MyClass[ ]> sp3(new MyClass[n]); // [] since C++17
    ...
} // proper delete called here
```

shared_ptr array specialization

- Array spec. calls delete[]
- Only [] operator, no * and ->
- No conversion from shared_ptr<Der[]> to shared_ptr<Base[]>

```
struct Base {
    virtual void f() { std::cout << "Base::f\n"; }
    ~Base() { std::cout << "Base::~~Base()\n"; }
};
struct Derived : Base {
    virtual void f() override { std::cout << "Derived::f\n"; }
    ~Derived() { std::cout << "Derived::~~Derived()\n"; }
};

int main() {
    std::shared_ptr<Derived[]> sp(new Derived[5]);
    std::shared_ptr<Base[]> bp(sp); // compile error
    auto p = sp[0];
    auto d0 = *sp; // compile error
}
```

shared_ptr

```
void f()
{
    std::shared_ptr<int> p1(new int{5});
    std::shared_ptr<int> p2 = p1; // now both own the memory.

    p1.reset(); // memory still exists, due to p2.
} // p2 out of scope: delete the memory, since no one else owns.
```

```
T* get() const noexcept;
T& operator*() const noexcept;
T* operator->() const noexcept;
T& operator[](idx) const noexcept; // returns get()[idx]
```

```
long use_count() const noexcept;
bool unique() const noexcept;
explicit operator bool() const noexcept;
```

weak_ptr

- Not owns the memory
- But part of the “sharing group”
- No direct operation to access the memory
- Can be converted to shared_ptr with lock()

```
long use_count() const noexcept;
bool expired() const noexcept;    // use_count() == 0

shared_ptr<T> lock() const noexcept;
// return expired() ? shared_ptr<T>() : shared_ptr<T>(*this)

void reset() noexcept;
```

Using lock()

```
void f()
{
    std::shared_ptr<X> ptr1 = std::make_shared<X>();
    std::shared_ptr<X> ptr2 = ptr1;

    std::weak_ptr<X> wptr = ptr2;

    if ( auto sp = wptr.lock() )
    {
        // use sp
    } // destructor of sp called here: release X object
    else
    {
        // expired
    }
} // destructor of X object is called here
```

Using lock()

```
int main ()
{
    std::shared_ptr<int> sp1, sp2;
    std::weak_ptr<int> wp;

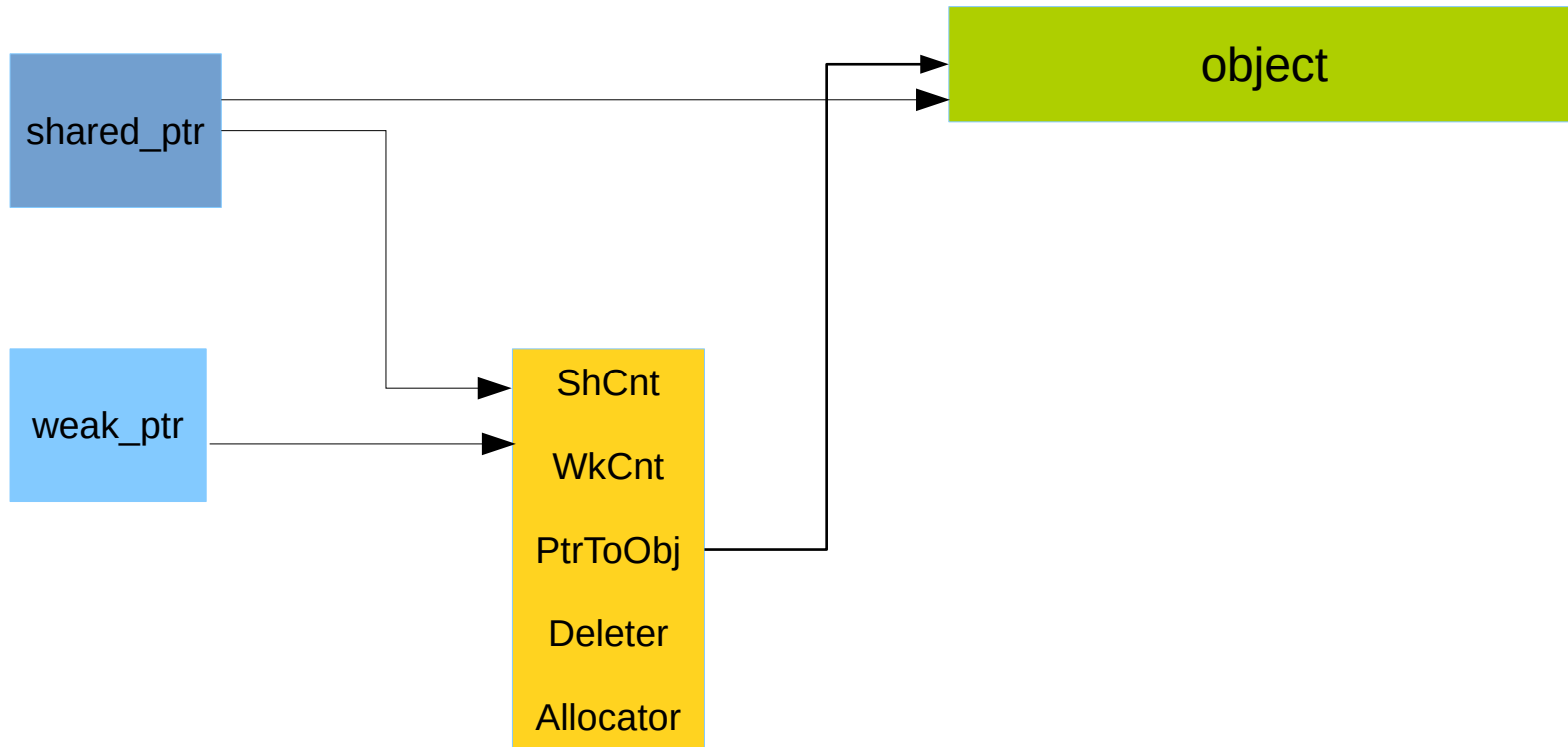
    // sharing group:
    // -----
    // sp1
    // sp1, wp

    sp1 = std::make_shared<int> (20);
    wp = sp1;

    sp2 = wp.lock();
    sp1.reset();
    // sp1, wp, sp2
    //      wp, sp2

    sp1 = wp.lock();
    // sp1, wp, sp2
}
```

Typical shared_ptr implementation



Enable shared from this

```
#include <memory>
#include <cassert>

class Y
{
public:

    std::shared_ptr<Y> f()
    {
        return shared_ptr<Y>(this); // ???
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p));
}
```

Enable shared from this

```
#include <memory>
#include <cassert>

class Y
{
public:

    std::shared_ptr<Y> f()    // BAD!!!
    {
        return shared_ptr<Y>(this);    // ???
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);          // failes
    assert(!(p < q || q < p));    // failes
}
```

Enable shared from this

```
#include <memory>
#include <cassert>

class Y
{
public:
    Y() : ptr_to_me(std::shared_ptr<Y>(this)) { }
    std::shared_ptr<Y> f()
    {
        return ptr_to_me; // ???
    }
private:
    std::shared_ptr<Y> ptr_to_me;
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p));
}
```

Enable shared from this

```
#include <memory>
#include <cassert>

class Y
{
public:
    Y() : ptr_to_me(std::shared_ptr<Y>(this)) { }
    std::shared_ptr<Y> f() // BAD!!!
    {
        return ptr_to_me; // ???
    }
private:
    std::shared_ptr<Y> ptr_to_me;
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p));
}
```

Enable shared from this

```
#include <memory>
#include <cassert>

class Y : public std::enable_shared_from_this<Y>
{
public:

    std::shared_ptr<Y> f() // OK
    {
        return shared_from_this();
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p)); // p and q share ownership
}
```

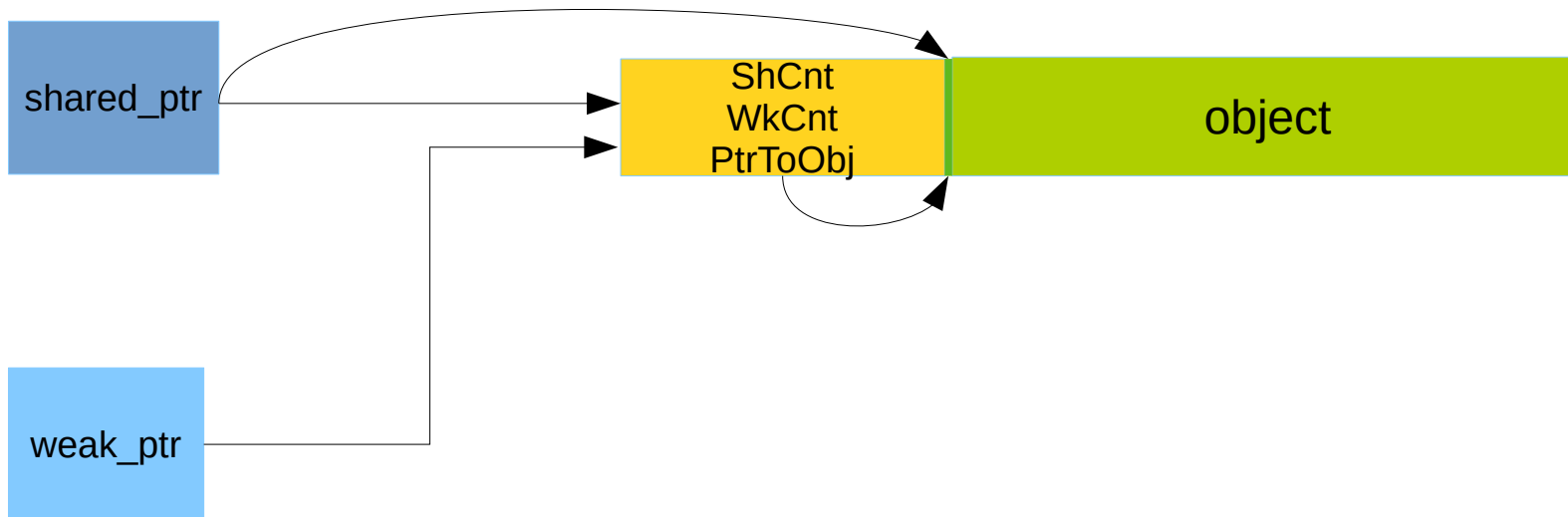
Make functions

```
// For unique_ptr
// default constructor of T
std::unique_ptr<T> v1 = std::make_unique<T>();
// constructor with params
std::unique_ptr<T> v2 = std::make_unique<T>(x,y,z);
// array of 5 elements
std::unique_ptr<T[]> v3 = std::make_unique<T[]>(5);

// similar methods for shared_ptr
```

Make functions

```
// For shared_ptr
// default constructor of T
std::shared_ptr<T> v1 = std::make_shared<T>();
// constructor with params
std::shared_ptr<T> v2 = std::make_shared<T>(x,y,z);
// array of 5 elements
std::shared_ptr<T[]> v3 = std::make_shared<T[]>(5);
```



Trap: exception safety

```
int f(); // may throw exception
```

```
// possible memory leak
```

```
std::pair<std::unique_ptr<MyClass>, int> foo()
```

```
{
```

```
    return std::make_pair(std::unique_ptr<MyClass>(new MyClass()), f());
```

```
}
```

Trap: exception safety

```
int f(); // may throw exception

// possible memory leak
std::pair<std::unique_ptr<MyClass>, int> foo()
{
    return std::make_pair(std::unique<MyClass>(new MyClass()), f());
}
```

1. Runs **new** MyClass
2. Runs f() and **throw** exception
3. std::unique_ptr constructor is not called

Trap: exception safety

```
int f(); // may throw exception
```

```
// possible memory leak
```

```
std::pair<std::unique_ptr<MyClass>, int> foo()  
{  
    return std::make_pair(std::unique<MyClass>(new MyClass()), f());  
}
```

```
int f(); // may throw exception
```

```
// safe
```

```
std::pair<std::unique_ptr<MyClass>, int> foo()  
{  
    return std::make_pair(std::make_unique<MyClass>(), f());  
}
```

Trap: exception safety

```
int f(); // may throw exception
```

```
// possible memory leak
```

```
std::pair<std::unique_ptr<MyClass>, int> foo()  
{  
    return std::make_pair(std::unique<MyClass>(new MyClass()), f());  
}
```

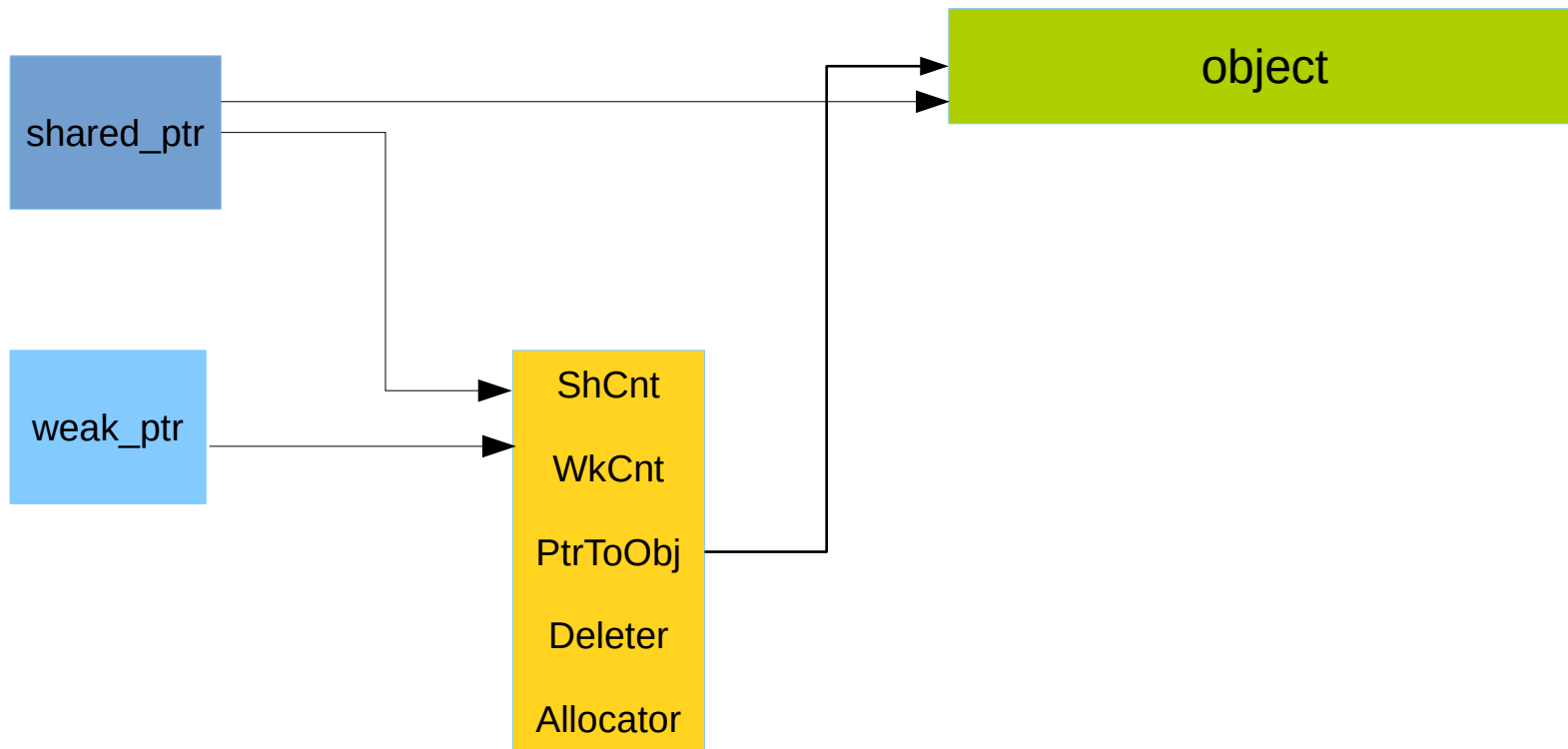
```
int f(); // may throw exception
```

```
// safe
```

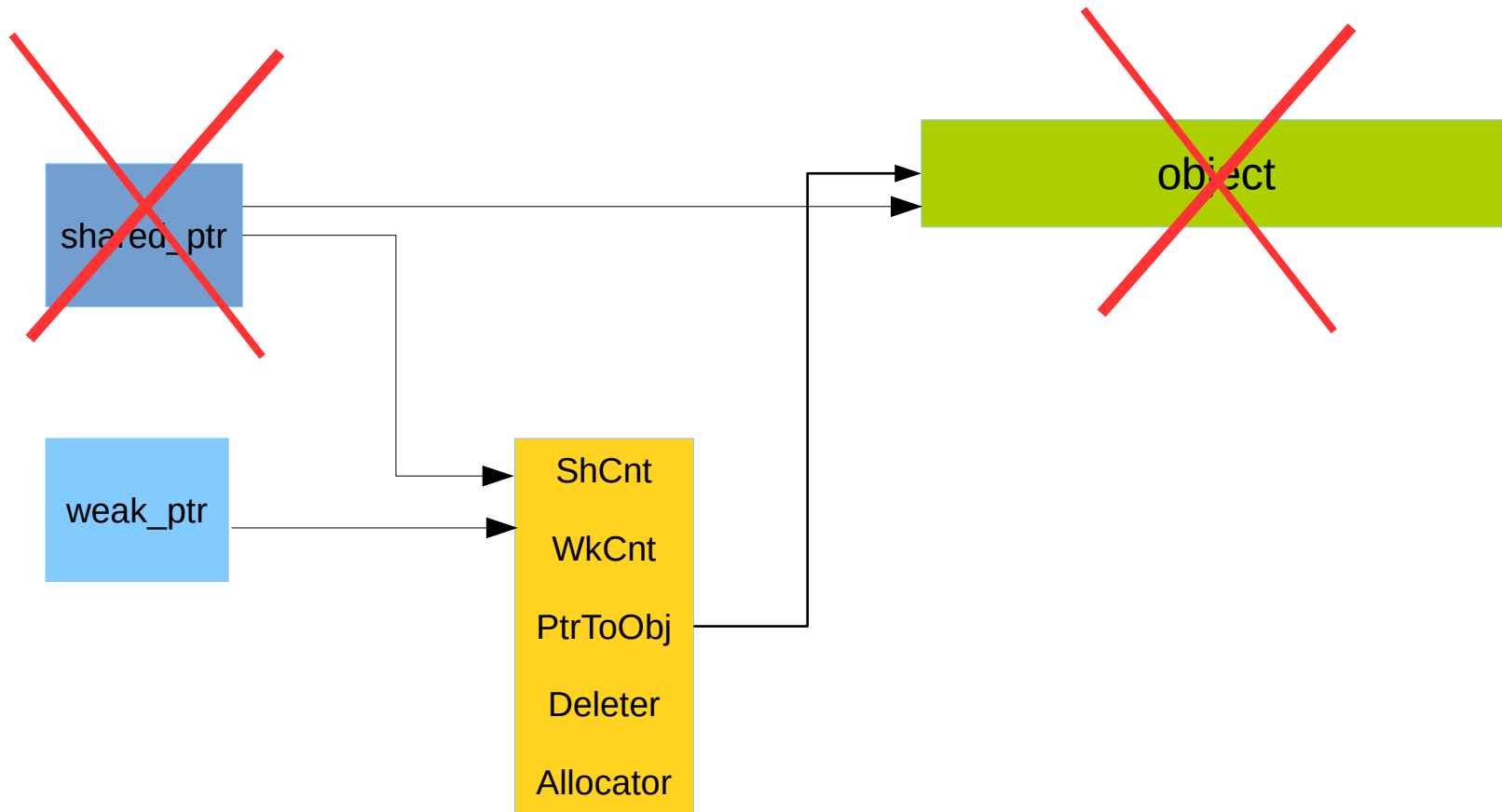
```
std::pair<std::unique_ptr<MyClass>, int> foo()  
{  
    return std::make_pair(std::make_unique<MyClass>(), f());  
}
```

No news – good news!

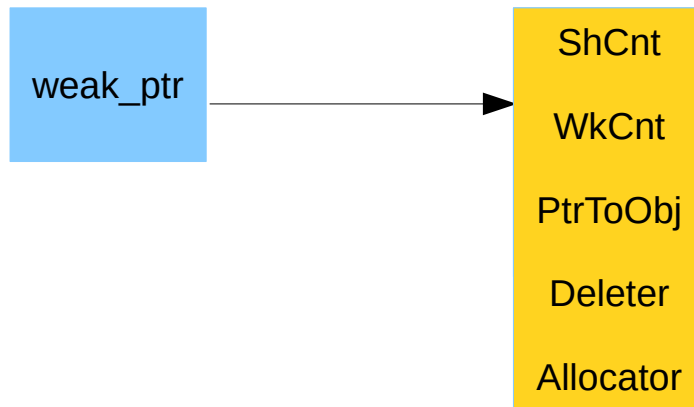
Trap: overuse of memory



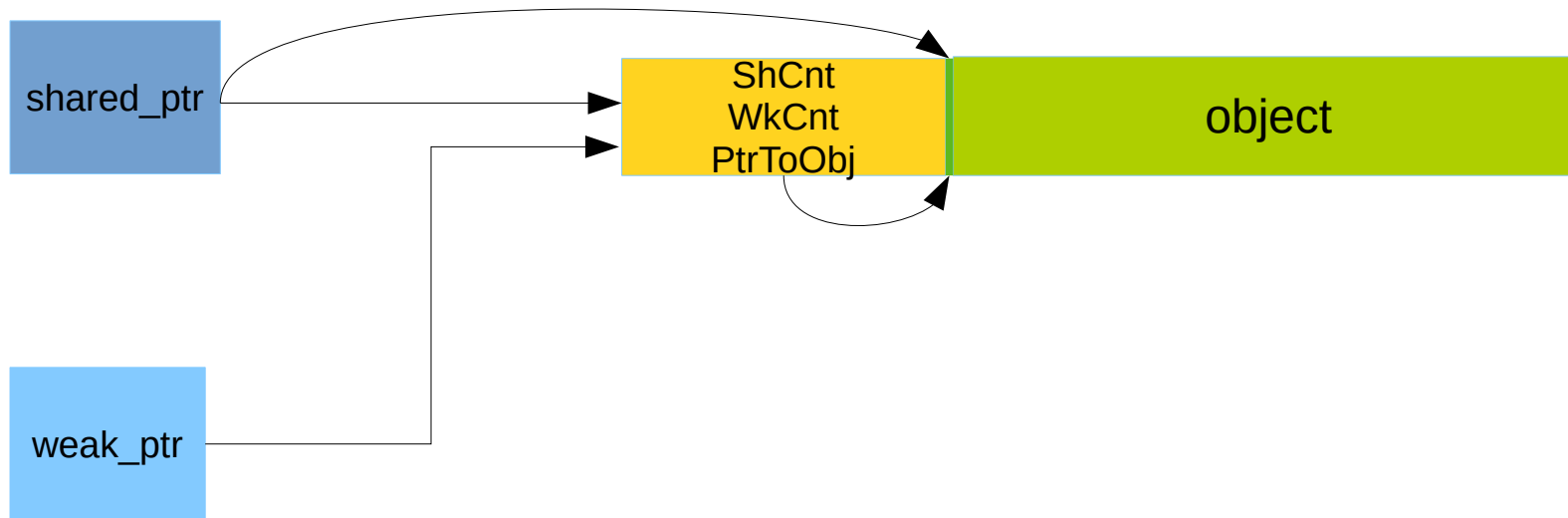
Trap: overuse of memory



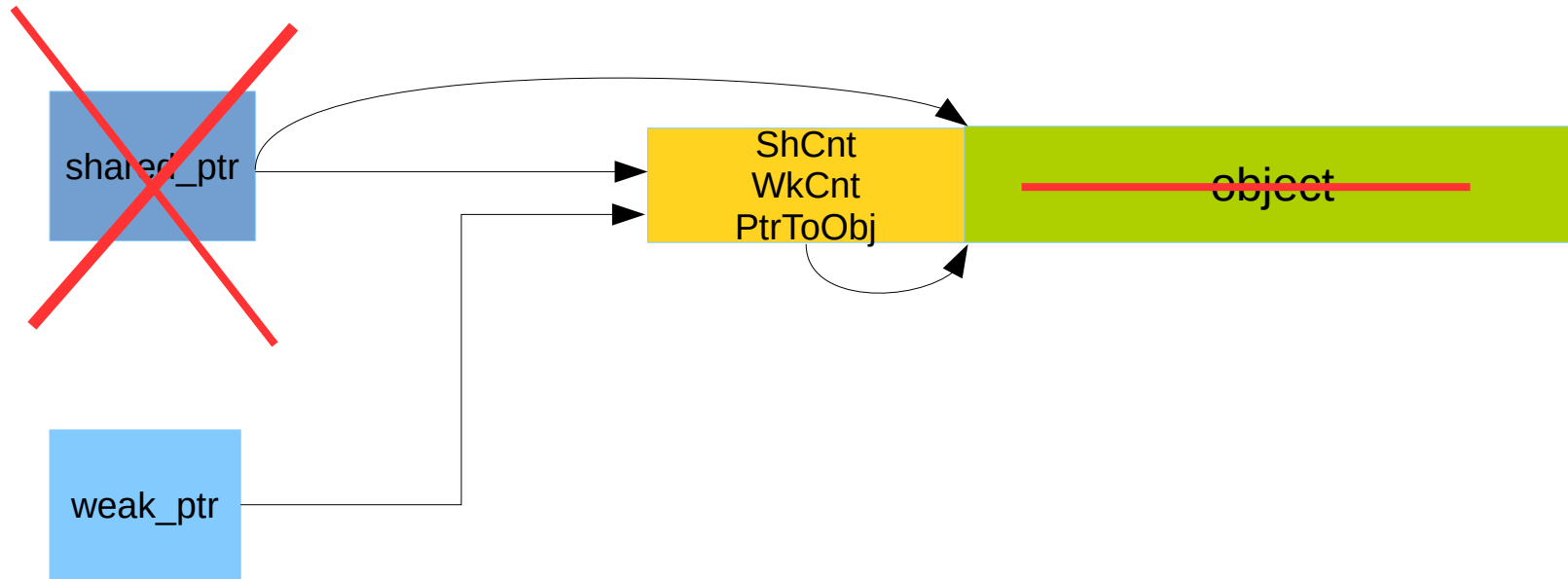
Trap: overuse of memory



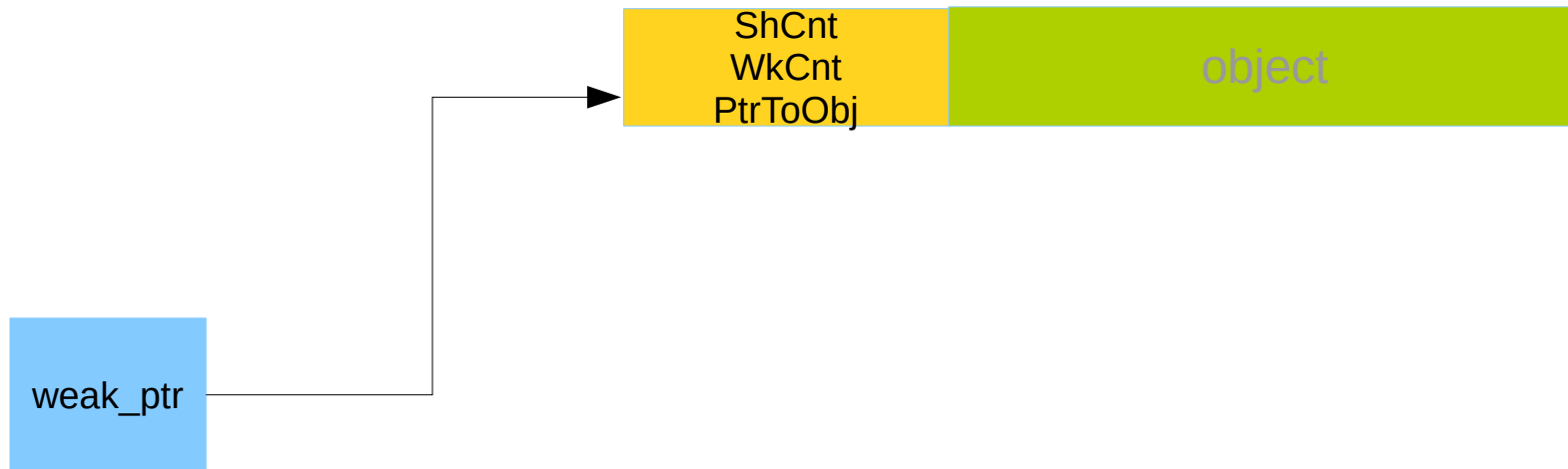
Trap: overuse of memory



Trap: overuse of memory



Trap: overuse of memory



When NOT to use `make_*`

- Both
 - You need custom deleter
 - You want to use braced initializer
- `std::unique_ptr`
 - You want custom allocator
- `std::shared_ptr`
 - Long living `weak_ptrs`
 - Class-specific `new` and `delete`
 - Potential false sharing of the object and the reference counter

Shared_ptr aliasing constructor

```
template< class Y >  
shared_ptr( const shared_ptr<Y>& r, element_type* ptr ) noexcept;
```

```
// since C++20
```

```
template< class Y >  
shared_ptr( shared_ptr<Y>&& r, element_type* ptr ) noexcept;
```

- Shares ownership with r
 - Reference counter is common with r
- Points to p
 - get() and -> returns p

Shared_ptr aliasing constructor

```
struct Data {
    int _i;
    Data( int i) : _i{i} { }
    virtual ~Data() { std::cout<<"Data::~Data(): "<<_i<<"\n"; }
};
struct Wrapper {
    Data _data;
    Wrapper(int i) : _data{i} { }
    ~Wrapper() { std::cout<<"Wrapper::~Wrapper(): "<<_data._i<<"\n"; }
};
int main()
{
    {
        Wrapper{1};
        const Data &dr = Wrapper{2}._data;
        std::cout << "end block\n";
    }
}
```

Shared_ptr aliasing constructor

```
struct Data {
    int _i;
    Data( int i) : _i{i} { }
    virtual ~Data() { std::cout<<"Data::~~Data(): " <<_i<<"\n"; }
};
struct Wrapper {
    Data _data;
    Wrapper(int i) : _data{i} { }
    ~Wrapper() { std::cout<<"Wrapper::~~Wrapper(): " <<_data._i<<"\n"; }
};
int main()
{
    {
        Wrapper{1};
        const Data &dr = Wrapper{2}._data; // lifetime extension
        std::cout << "end block\n";
    }
}
```

```
Wrapper::~~Wrapper(): 1
Data::~~Data(): 1
end block
Wrapper::~~Wrapper(): 2
Data::~~Data(): 2
```

Shared_ptr aliasing constructor

```
struct Data {
    int _i;
    Data( int i) : _i{i} { }
    virtual ~Data() { std::cout<<"Data::~~Data(): "<<_i<<"\n"; }
};
struct Wrapper {
    Data _data;
    Wrapper(int i) : _data{i} { }
    ~Wrapper() { std::cout<<"Wrapper::~~Wrapper(): "<<_data._i<<"\n"; }
};
int main()
{
    {
        std::shared_ptr<Wrapper> wp = std::make_shared<Wrapper>(1);
        std::shared_ptr<Data> dp(wp, &wp->_data);
        wp.reset();
        std::cout << dp->_i << '\n';
        std::cout << "end block\n";
    }
}
```

```
1
end block
Wrapper::~~Wrapper(): 1
Data::~~Data(): 1
```

for_overwrite (C++20)

```
auto ap1 = std::make_unique<int[]>(1000); // creates 1000 int  
  
for ( auto i = 0; i < 1000; ++i)  
    ap1[i] = i; // overwrites the elements
```

for_overwrite (C++20)

```
auto ap1 = std::make_unique<int[]>(1000); // creates 1000 int
                                           // and value initialize each to 0
for ( auto i = 0; i < 1000; ++i)
    ap1[i] = i; // overwrites the elements
```

for_overwrite (C++20)

```
auto ap1 = std::make_unique<int[]>(1000); // creates 1000 int
                                           // and value initialize each to 0
for ( auto i = 0; i < 1000; ++i)
    ap1[i] = i; // overwrites the elements

auto ap2 = std::make_unique_for_overwrite<int[]>(1000); // creates 1000 int
                                                       // elements are default initialized
for ( auto i = 0; i < 1000; ++i)
    ap2[i] = i; // overwrites the elements

// similarly for shared_ptr
auto ap3 = std::make_shared_for_overwrite<int[]>(1000); // creates 1000 int
                                                       // elements are default initialized
```

for_overwrite (C++20)

```
auto ap1 = std::make_unique<int[]>(1000); // creates 1000 int
                                           // and initialize each to 0
for ( auto i = 0; i < 1000; ++i)
    ap1[i] = i; // overwrites the elements

auto ap2 = std::make_unique_for_overwrite<int[]>(1000); // creates 1000 int
                                                       // elements are default initialized
for ( auto i = 0; i < 1000; ++i)
    ap2[i] = i; // overwrites the elements

// similarly for shared_ptr
auto ap3 = std::make_shared_for_overwrite<int[]>(1000); // creates 1000 int
                                                       // elements are default initialized

auto ap4 = std::make_unique_for_overwrite<int[]>(1000); // creates 1000 int
++ap4[4]; // undefined behavior
```

allocate_shared (C++20)

- Use the allocator's alloc to allocate memory (instead of ::new)
- Constructor called by std::allocator_traits<A2>::construct(a, pv, v)

```
template <class T, class Alloc, class... Args>
shared_ptr<T> allocate_shared (const Alloc& alloc, Args&&... args);

#include <iostream>
#include <memory>

void f()
{
    std::allocator<int> alloc; // the default allocator for int
    std::default_delete<int> del; // the default deleter for int

    std::shared_ptr<int> foo = std::allocate_shared<int> (alloc, 10);
    auto bar = std::allocate_shared<int> (alloc, 20);
    auto baz = std::allocate_shared<std::pair<int, int>> (alloc, 30, 40);

    std::cout << "*foo: " << *foo << '\n';
    std::cout << "*bar: " << *bar << '\n';
    std::cout << "*baz: " << baz->first << ' ' << baz->second << '\n';
}
```

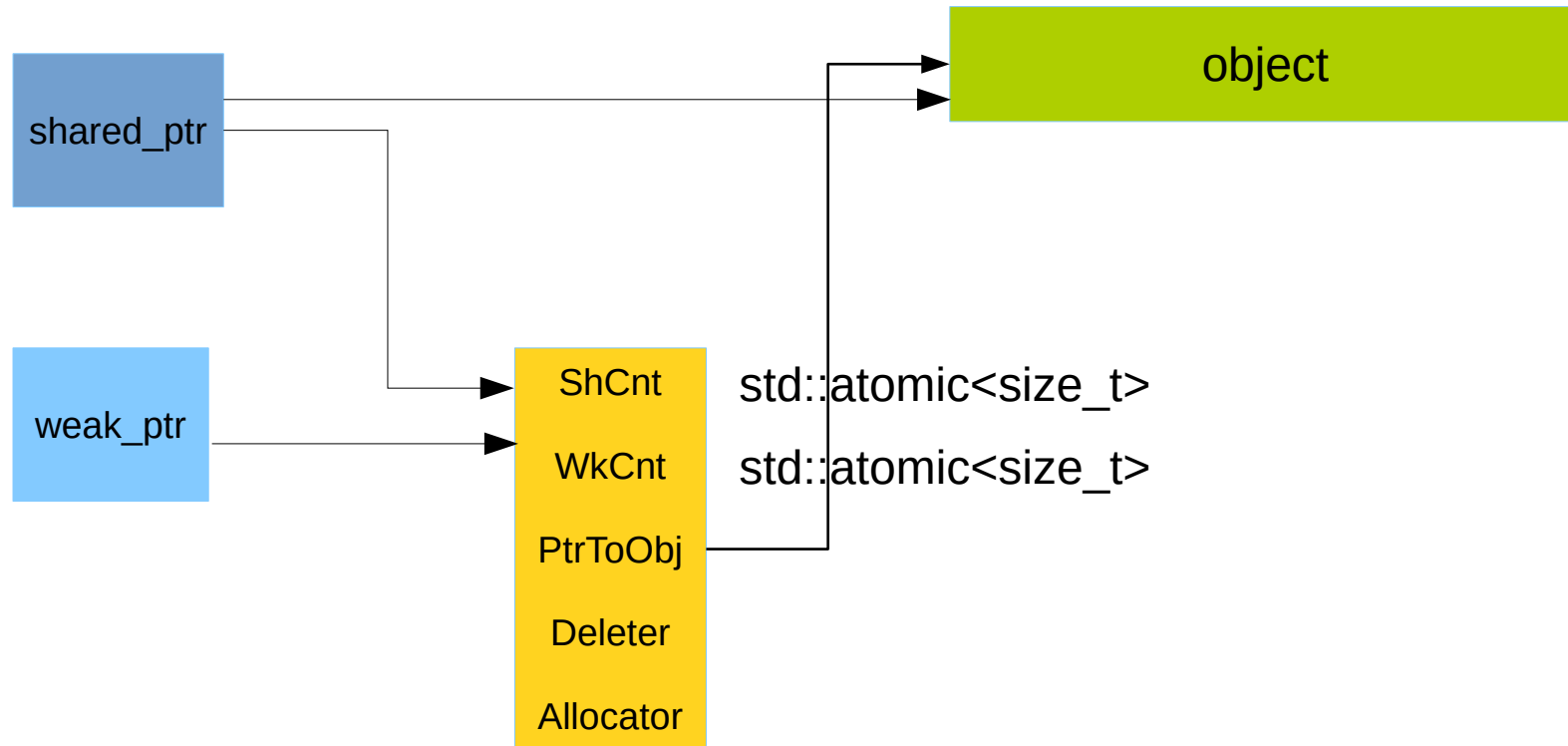
atomic<shared_ptr> (C++20)

- shared_ptr is not atomic and **thread** safe by **default**
- std::experimental::atomic_shared_ptr merged into std::atomic<std::shared_ptr>
- Supported g++ 12, MSVC 19.27
- shared_ptr atomic operations were used before **auto**<shared_ptr>

```
template <class T>  
std::experimental::atomic_shared_ptr<T> // before C++20
```

```
template <class T>  
std::atomic<std::shared_ptr<T>> // since C++20
```

Shared_ptr implementation



atomic<shared_ptr> (C++20)

```
std::shared_ptr<X> ptr = std::make_shared<X>();
```

```
std::shared_ptr<X> ptr1 = ptr;           // ok, thread safe
```

thread1

thread2

```
std::shared_ptr<X> ptr2 = ptr;           // ok, thread safe
```

atomic<shared_ptr> (C++20)

```
std::shared_ptr<X> ptr = std::make_shared<X>();
```

```
std::shared_ptr<X> ptr1 = ptr;           // ok, thread safe
```

thread1

thread2

```
std::shared_ptr<X> ptr2 = ptr;           // ok, thread safe
```

```
ptr = ptr3;                             // data race, undefined behavior
```

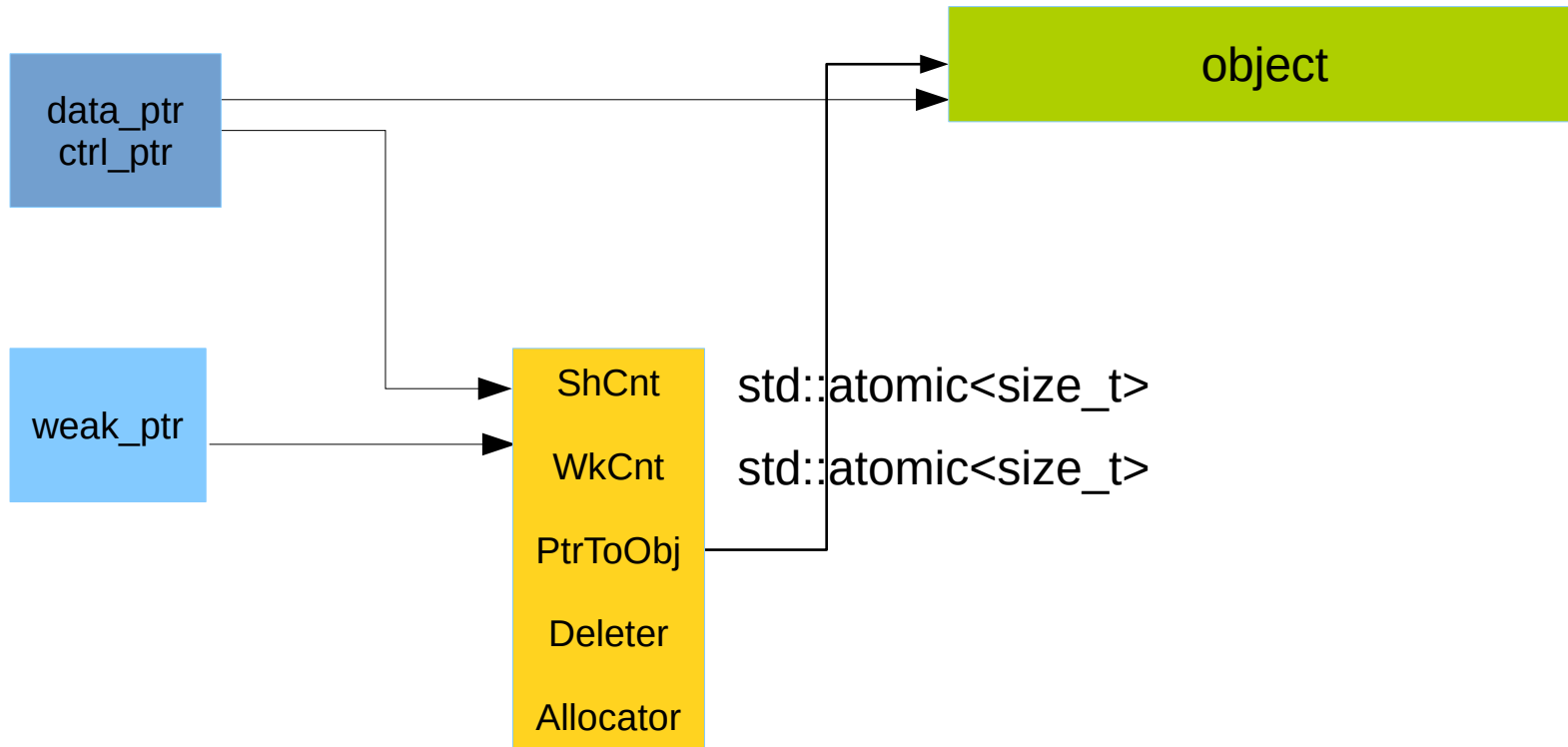
atomic<shared_ptr> (C++20)

```
std::shared_ptr<X> ptr = std::make_shared<X>();
std::atomic<std::shared_ptr<X>> aptr = std::make_shared<X>();

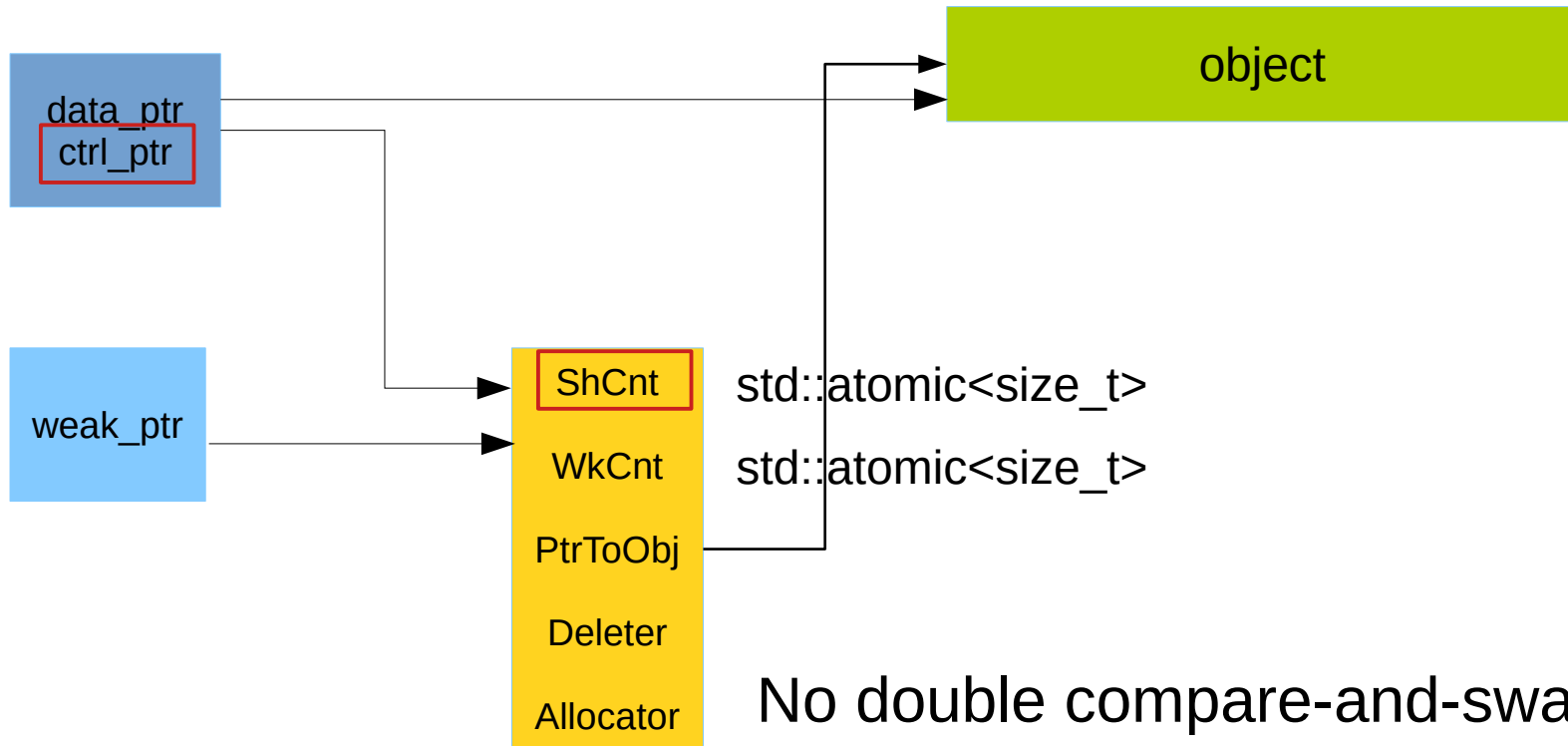
std::shared_ptr<X> ptr1 = ptr;           // ok, thread safe
std::atomic<std::shared_ptr<X>> aptr1 = aptr.load(); // atomic operation
                                                    thread1
-----
                                                    thread2
std::atomic<std::shared_ptr<X>> aptr2 = aptr; // ok, thread safe

ptr = ptr3;           // data race, undefined behavior
aptr.store(ptr3);    // ok, atomic operation
```

Shared_ptr implementation lock-free?



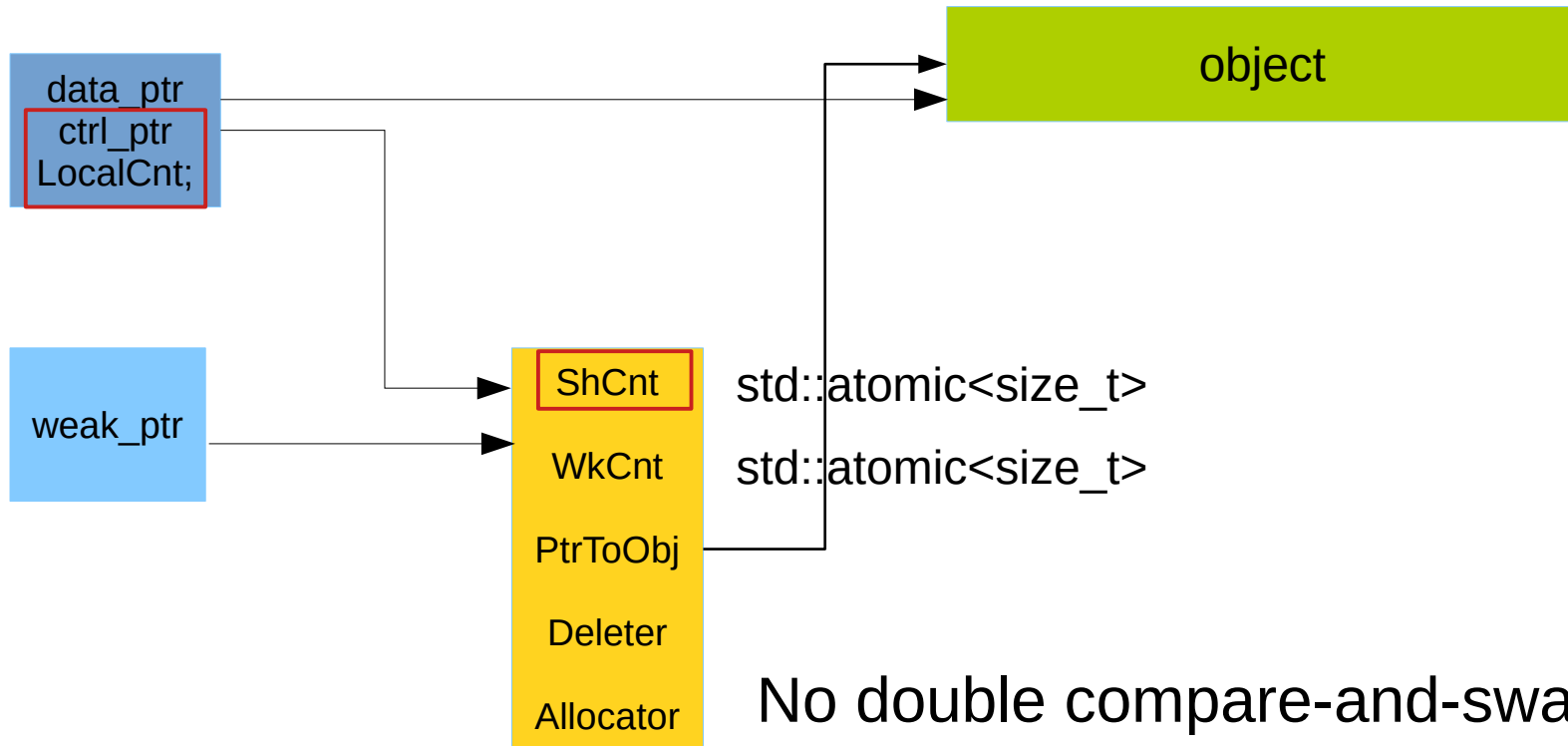
Shared_ptr implementation lock-free?



No double compare-and-swap

There is double-with compare-and-swap

Shared_ptr implementation lock-free?



No double compare-and-swap

There is double-with compare-and-swap

atomic<shared_ptr> (C++20)

```
template<typename T>
class atomic<shared_ptr<T>>
{
    struct counter_ptr {
        ControlBlk*   ctl_ptr;
        size_t        localCnt;
    };
    atomic<counter_ptr> cptr;
    static_assert(decltype(cptr)::is_always_lock_free);

    shared_ptr<T> load()
    {
        auto cptr_copy = cptr.load();
        while (true) {
            auto cptr_new = cptr_copy;
            ++cptr_new.localCnt;
            if ( cptr.compare_exchange_weak(cptr_copy, cptr_new) )
                break;
        }
        ++cptr_copy.localCnt;
        auto ctl_ptr = cptr_copy.ctl_ptr;
    }
};
```