

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

Typical RAII solutions

- Smart pointers for memory handling
- Guards for locking
- ifstream, ofstream objects for file-i/o
- std::containers

```
class X
{
public:
    void *non_thread_safe();
private:
    Mutex lock_;
};

void *X::non_thread_safe();
{
    Guard<Mutex> guard(lock_);
    /* critical section */
}
```

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

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 **Don't use it!**

```
int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;
    // p: 42    q: NULL

    q = p;
    // p: NULL q: 42

    *q += 13;    // change value of the object q owns
    p = q;
    // p: 55    q: NULL
}
```

How inheritance works?

- Raw pointers: assign `Derived*` to `Base*` works
- But `auto_ptr<Derived>` is not inherited from `auto_ptr<Base>` !
- We need a bit of trick here

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

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

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


Unique_ptr

- Single ownership pointer (similar to auto_ptr)
- But carefully designed to work with STL and other language features
- Movable but not copyable
- Deleter type parameter

```
#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); // syntax error: can't copy unique_ptr

    std::unique_ptr<Foo> up3; // nullptr: not owner

    up3 = up; // syntax error: can't assign unique_ptr
    up3 = std::move(up); // ownership moved to up3
} // up3 destroyed: Foo object is destructed
  // up destroyed: nop
```

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

- The `sizeof(unique_ptr<>)` without deleter is `== sizeof(raw pointer)`
- If there is deleter with state, the size increases
- If no state, then no size penalty (e.g. lambda with no capture).
- If no state, but function pointer is used as deleter: `sizeof(funptr)` added
- Prefer `unique_ptr` when possible
- Can be used in standard containers when polymorphic use needed
- Cheap
- No downcast operation :(

shared_ptr

- Shared ownership pointer with reference counter
- Copy constructible and assignable
- No array specializations (e.g. no `shared_ptr<T[]>`)
- Deleter type parameter

```
#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[]>());
    ...
} // proper delete called here
```

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.reset(); // delete the memory, since no one else owns.
}
```

```
T & operator*() const; // never throws
T * operator->() const; // never throws
T * get() const; // never throws

bool unique() const; // never throws
long use_count() const; // never throws
```

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;
bool expired() const;    // use_count() == 0

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

void reset();
```


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
    else
        // expired
} // destructors are 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;

    // sp1, wp, sp2
    //      wp, sp2

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

    // sp1, wp, sp2

    sp1 = wp.lock();
}
```

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);

// For shared_ptr
void f()
{
    std::unique_ptr<Foo> up(new Foo()); // up is the only owner

    if ( up ) // owner or not
    {
        *up = ...; // use the object
    }
}
```

Enable shared from this

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

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

    std::shared_ptr<Y> f()
    {
        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 must share ownership
}
```

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());
}
```

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());  
}
```

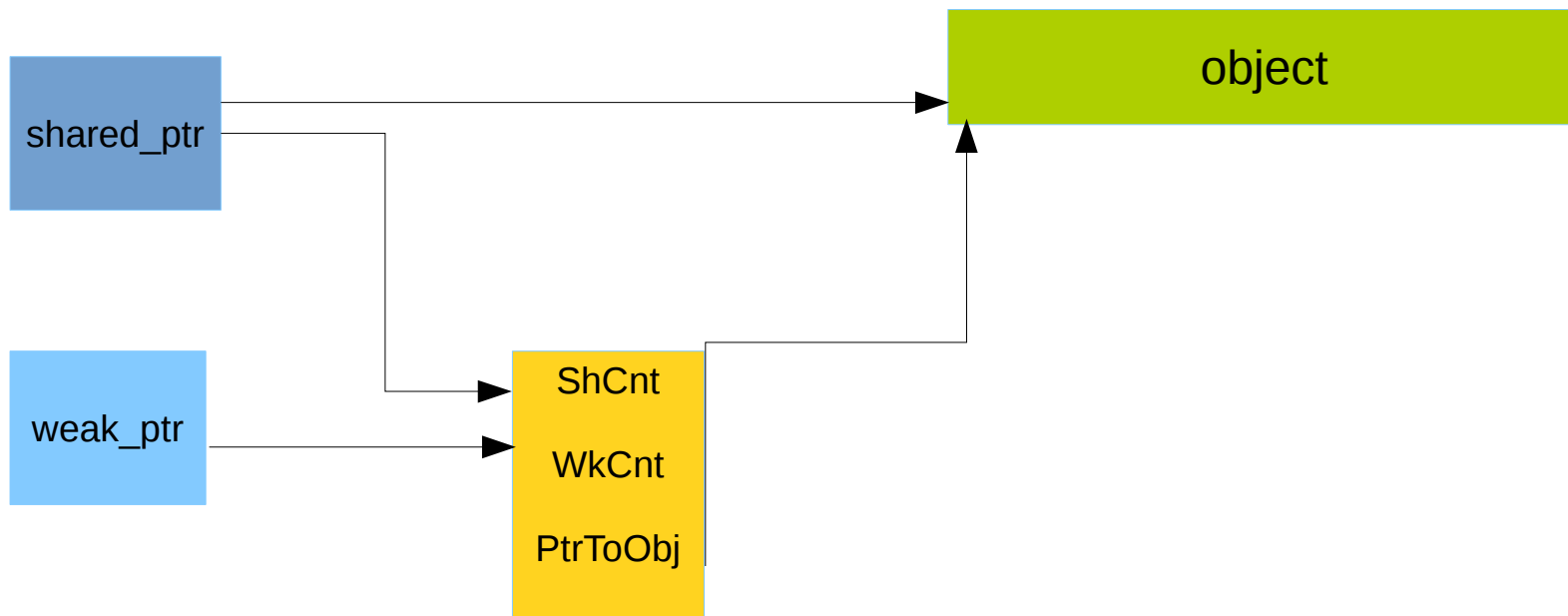
```
int f(); // may throw exception
```

```
// safe
```

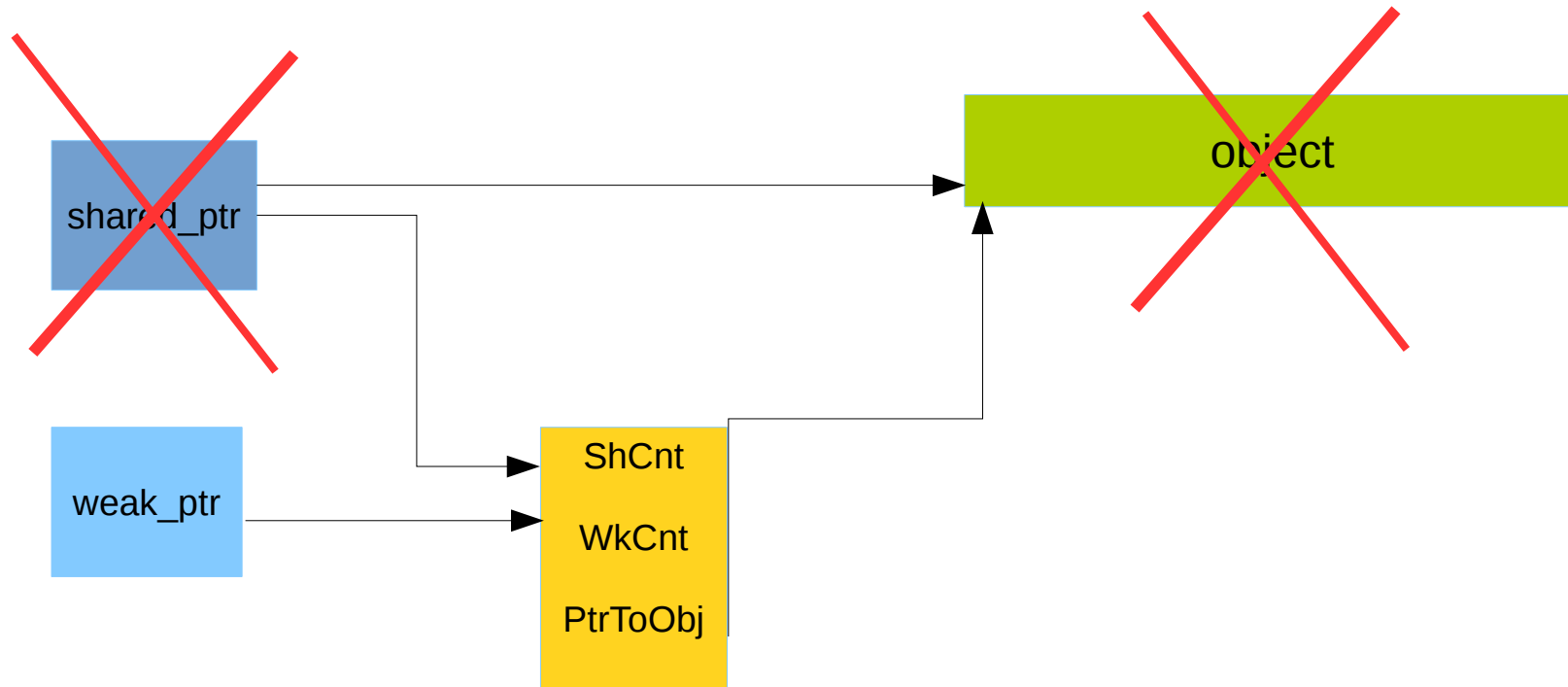
```
std::pair<std::unique_ptr<MyClass>,int> foo()  
{  
    return std::make_pair(std::make_unique<MyClass>(), f());  
}
```

No news – good news!

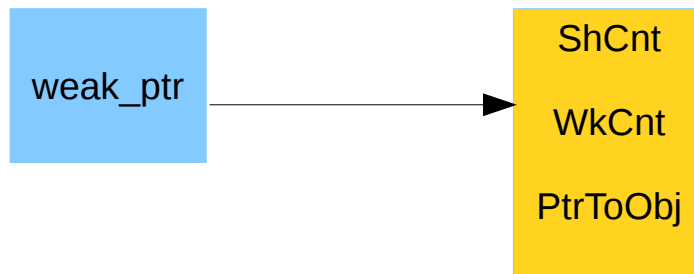
Trap: overuse of memory



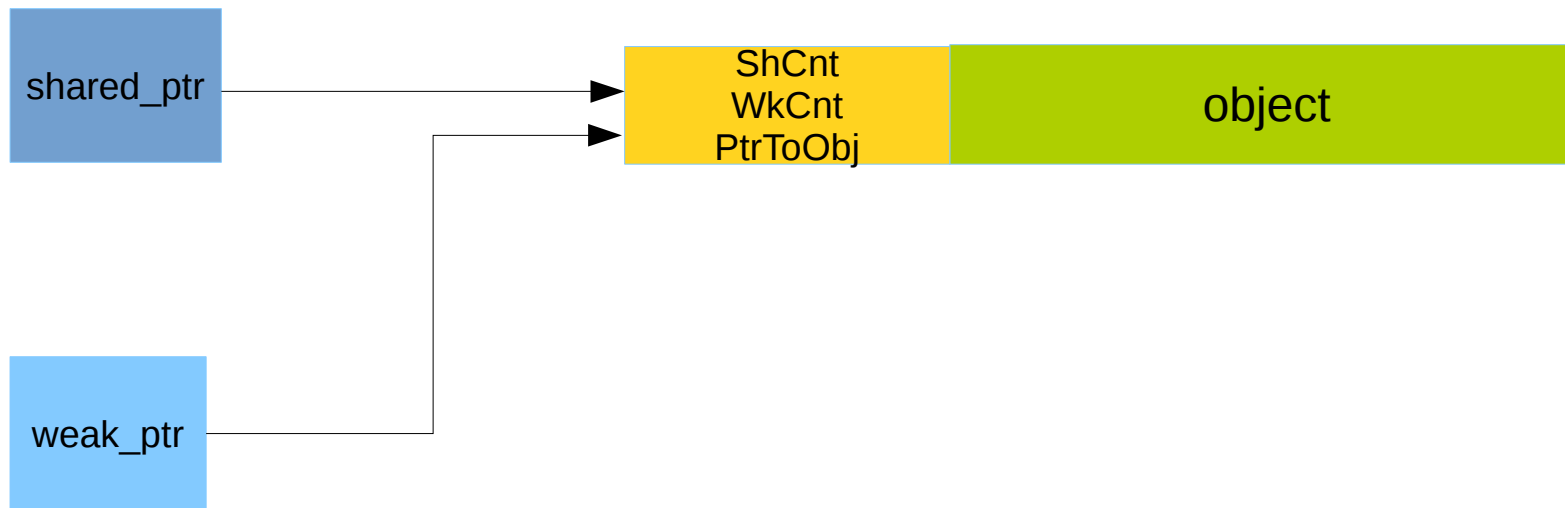
Trap: overuse of memory



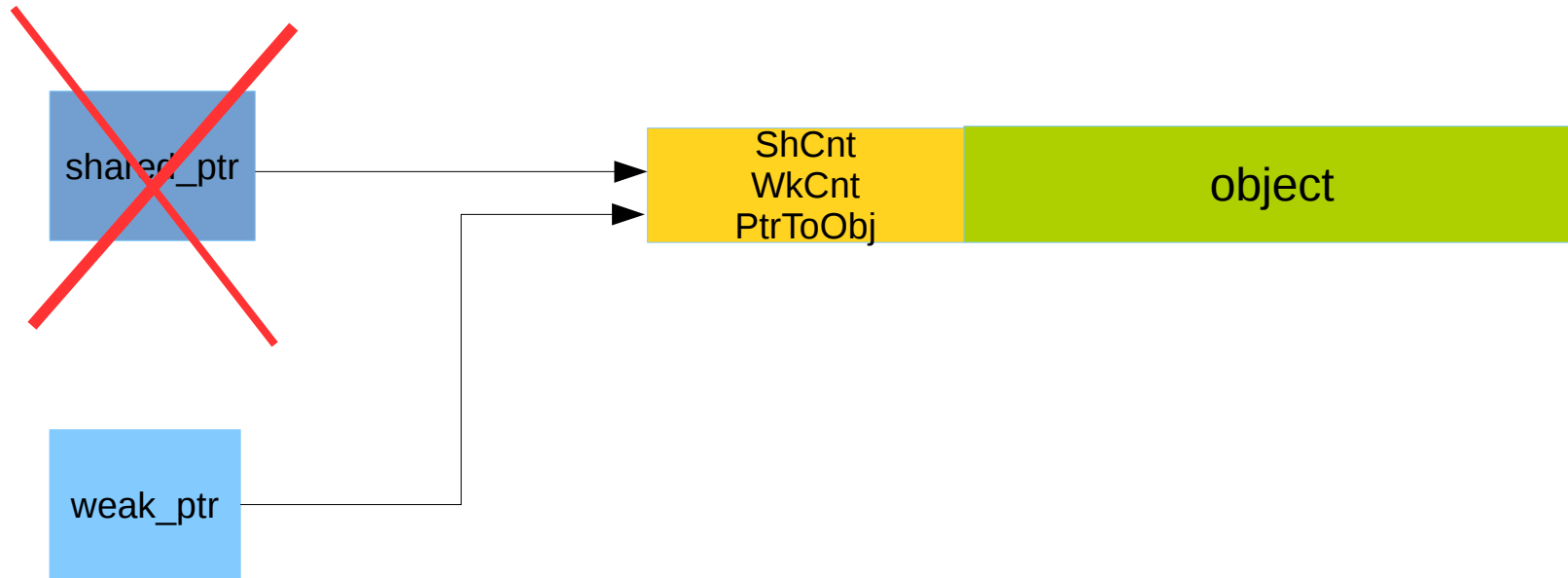
Trap: overuse of memory



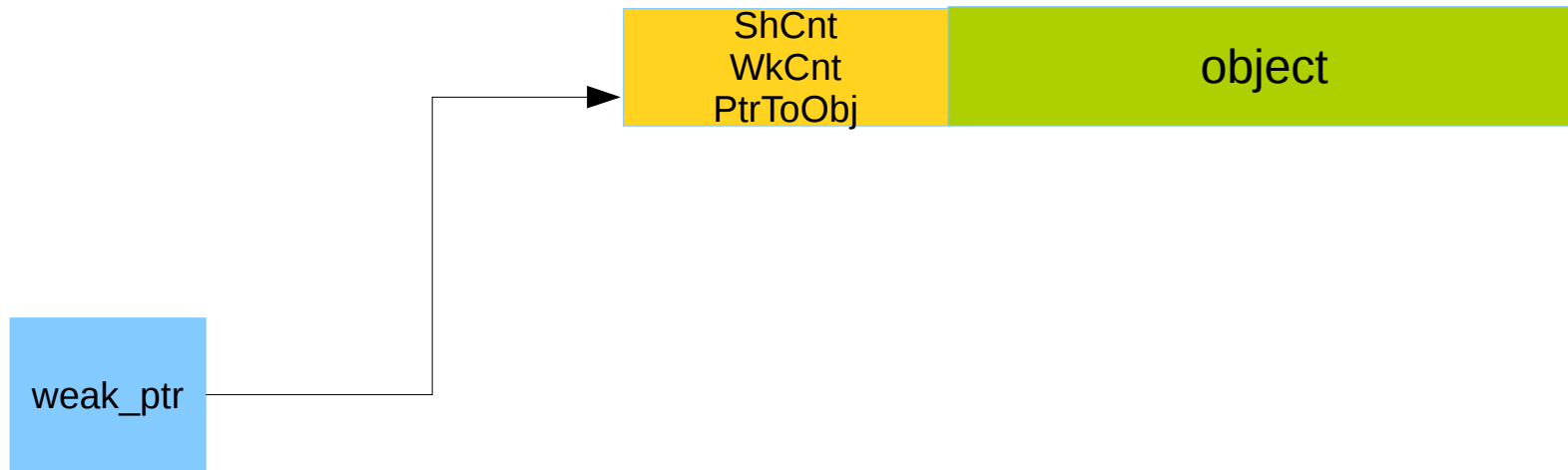
Trap: overuse of memory



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