[CS 225] Advanced C/C++

Lecture 15: Smart pointers and exceptions

Sławomir "Swavek" Włodkowski
(Fall 2020)

Agenda

- Smart pointers
- Review of exceptions
- Exception safety guarantees

Smart pointers

An abstraction (`<memory>`) for managing a free store object (you won't forget to `delete` ever again!):

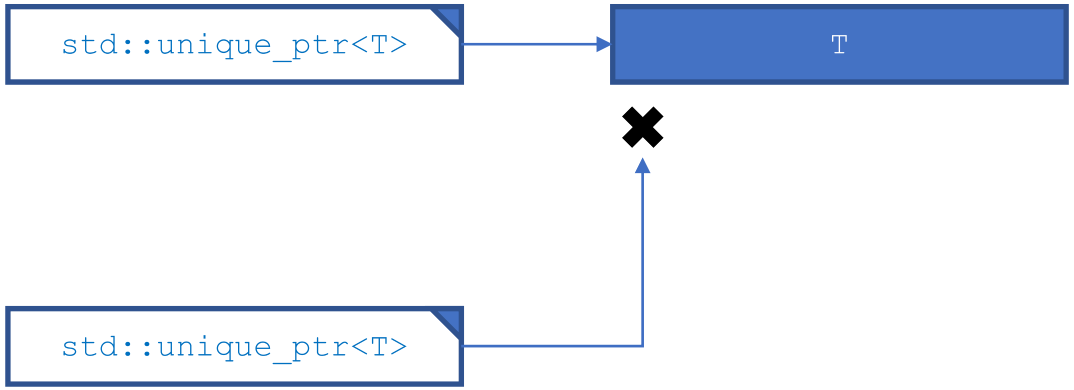
- `std::unique_ptr` – single owner of a dynamic object (movable, not copiable); when destroyed destroys the object .
- `std::shared_ptr` – each copy becomes co-owner of a dynamic object; last destroyed copy destroys the object.
- `std::weak_ptr` – created from `std::shared_ptr`; if the object has not been yet destroyed, it can be converted back.

Smart pointers

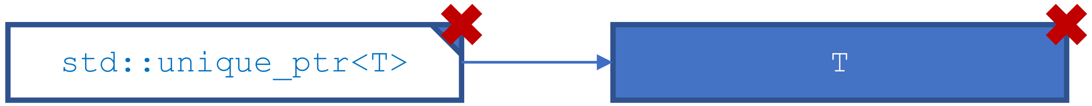
There are other smart pointers:

- `std::auto_ptr` – pre-C++11 attempt at unique pointers; unsuccessful due to lack of move semantics. Do not use.
- System-specific (COM, ATL) – useful for OS integration.

std::unique_ptr



std::unique_ptr



std::unique_ptr

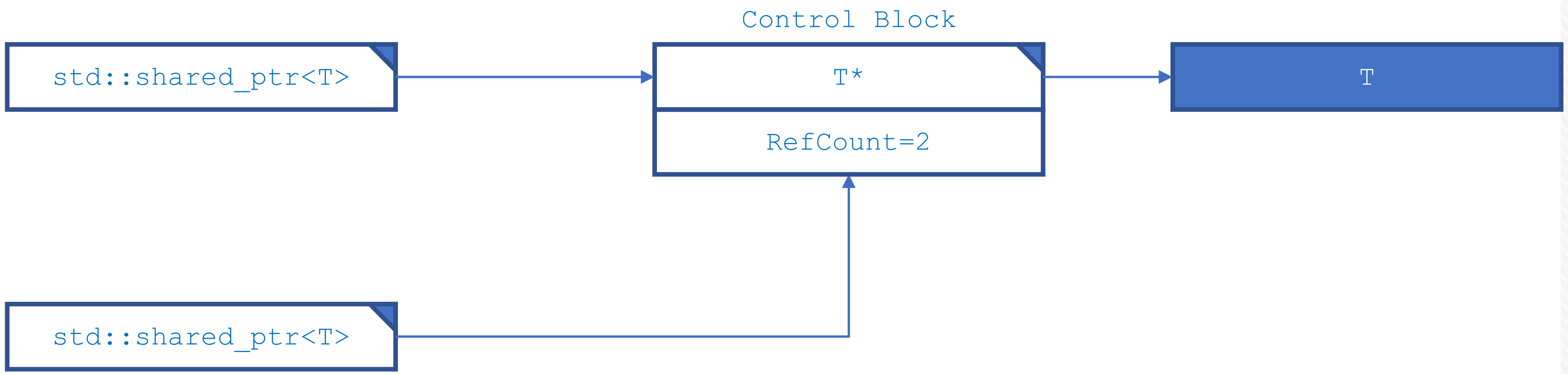
```cpp
#include <memory>
#include <iostream>

struct MyClass {
    void print() const { std::cout << "Hello world!" << std::endl; }
};

std::unique_ptr<MyClass> create() { return std::make_unique<MyClass>(/* c-tor params */); }

void print1(MyClass& obj) { obj.print(); }

template <typename T>
void print2(T&& obj_ptr) { std::forward<T>(obj_ptr)->print(); }

void print3(std::unique_ptr<MyClass> obj) { obj->print(); }

int main() {
    std::unique_ptr<MyClass> obj = create();
    print1(*obj);              // Dereference
    print2(obj.get());         // Pointer
    print2(obj);               // Smart pointer
    print3(std::move(obj));    // Moved smart pointer
}
```

`std::unique_ptr`

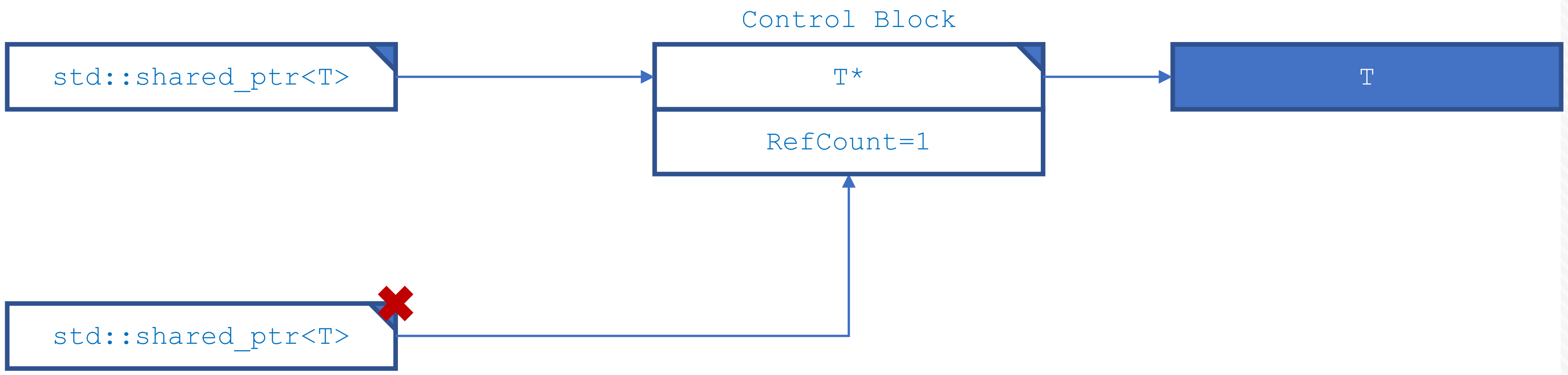
- Creating
  - Using a constructor
    `std::unique_ptr<T>{new T{/*params*/}}`
  - Or even better, without using `new`:
    `std::make_unique<T>(/*params*/)`

- Using
  - `if (ptr) // checks for nullptr`
  - Offers array specialization with `operator[]` and proper `delete[]`.
  - Allows for a default or a custom deleter.

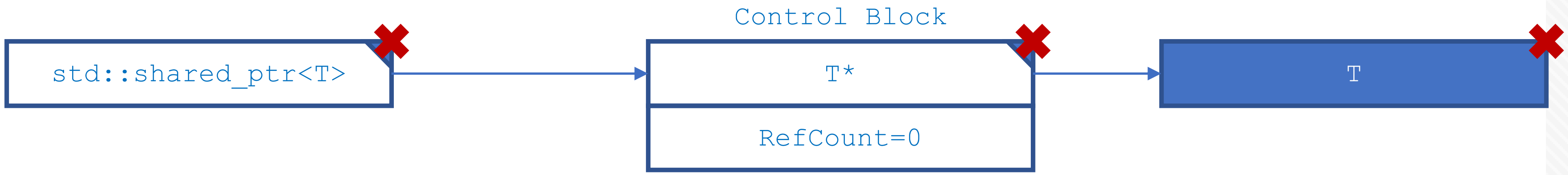
std::shared_ptr



std::shared_ptr



std::shared_ptr



std::shared_ptr

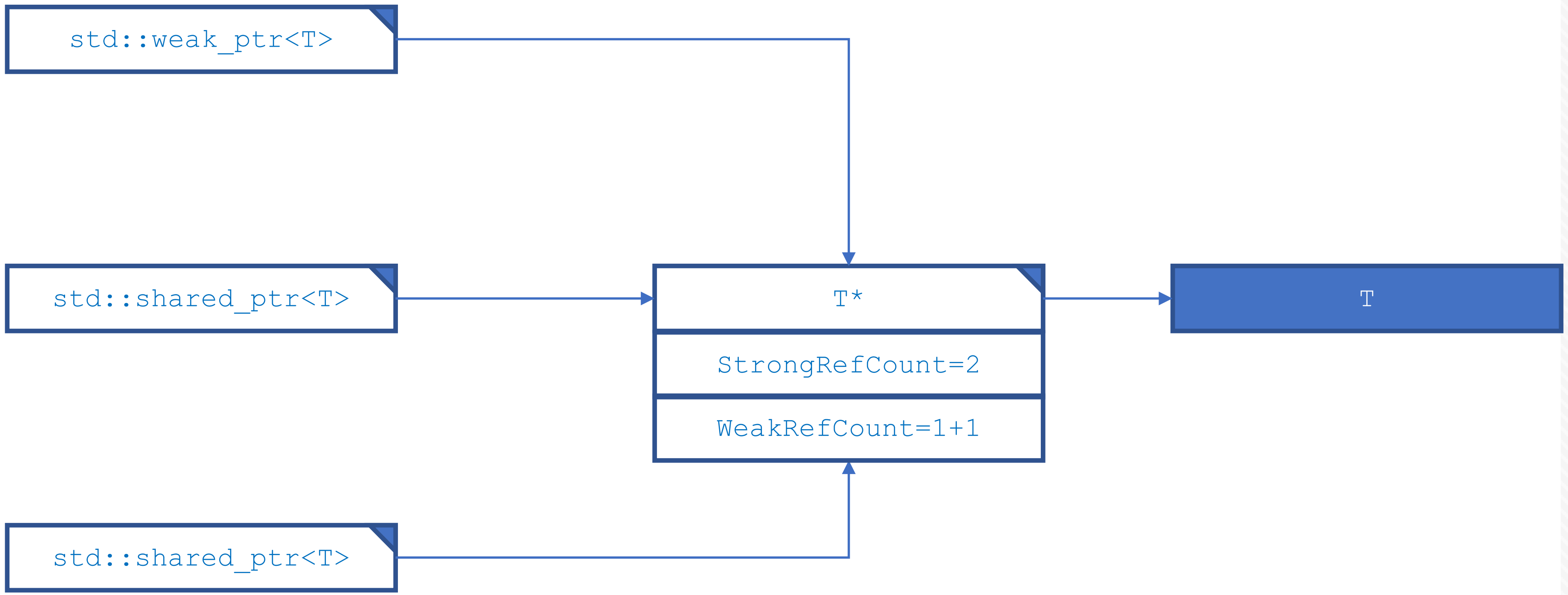
```
#include <memory>
#include <iostream>

struct MyClass {
    void print() const { std::cout << "Hello world!" << std::endl; }
};

std::shared_ptr<MyClass> create() { return std::make_shared<MyClass>(/* c-tor params */); }

void print1(MyClass& obj) { obj.print(); }

template <typename T>
void print2(T&& obj_ptr) { std::forward<T>(obj_ptr)->print(); }

void print3(std::shared_ptr<MyClass> obj) { obj->print(); }

int main() {
    std::shared_ptr<MyClass> obj = create();
    print1(*obj);              // Dereference
    print2(obj.get());         // Pointer
    print2(obj);               // Smart pointer
    print3(obj);               // Copied as a shared pointer
}
```

`std::shared_ptr`

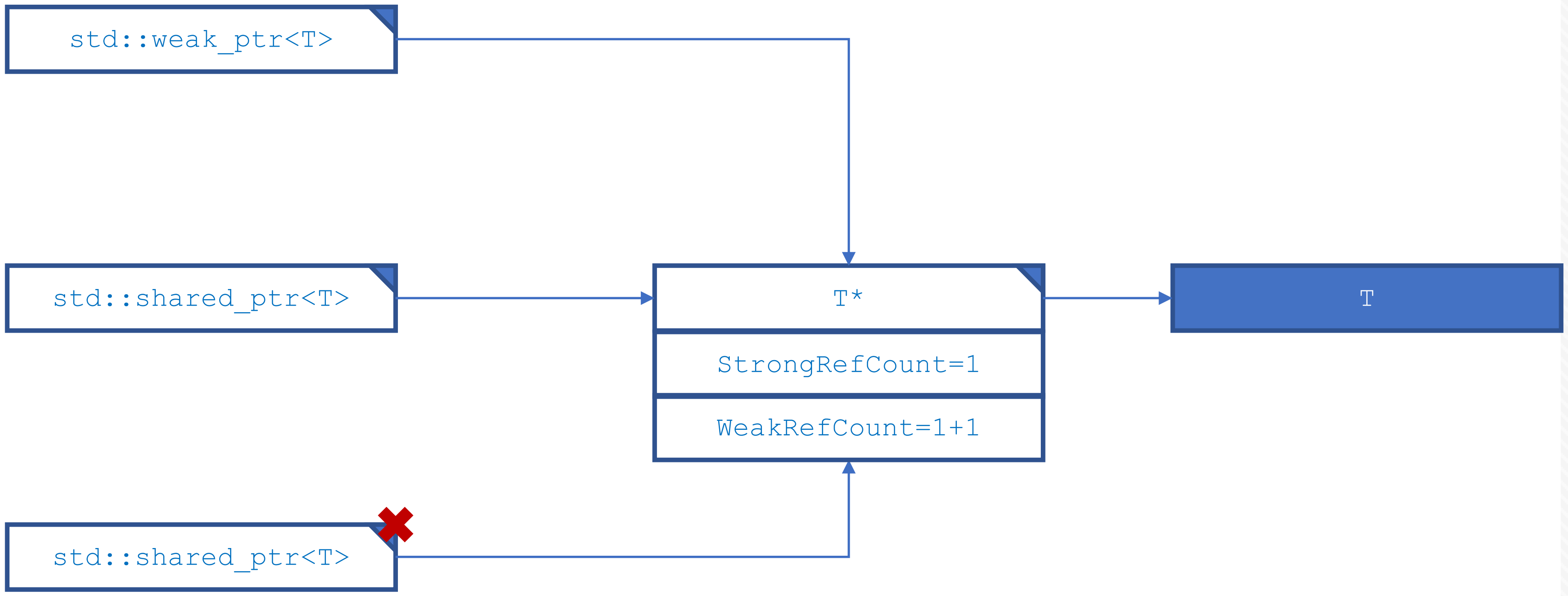
- Creating
  - Using a constructor
    `std::shared_ptr<T>{new T{/*params*/}}`
  - Or even better, without using `new`:
    `std::make_shared<T>(/*params*/)`
  - Sharing ownership by copy semantics (including passing by-value)
    `std::shared_ptr<T> s1 = s2;`

- Using
  - `if (ptr) // checks for nullptr`
  - Offers `operator[]`.
  - Offers `use_count()` exposing its reference counting mechanism.
  - Allows for a pointer to the object and a managed pointer to be different (*think: managing an object but pointing to its base class*).

std::weak_ptr

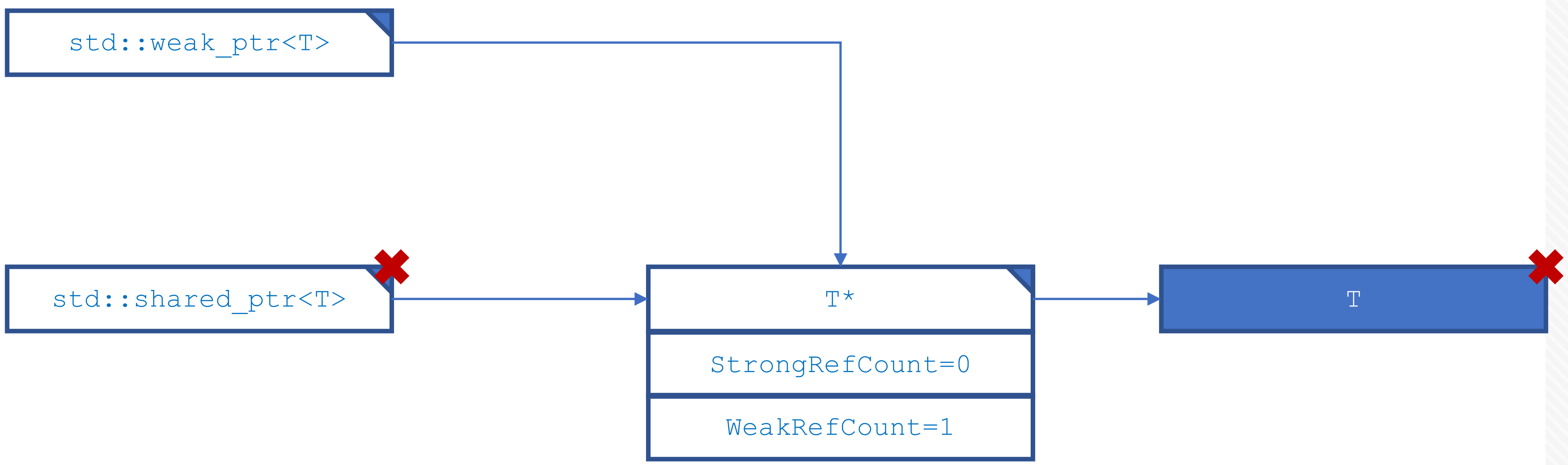

std::weak_ptr<T>
std::shared_ptr<T>
T*
StrongRefCount=2
WeakRefCount=1+1
T
std::shared_ptr<T>

std::weak_ptr

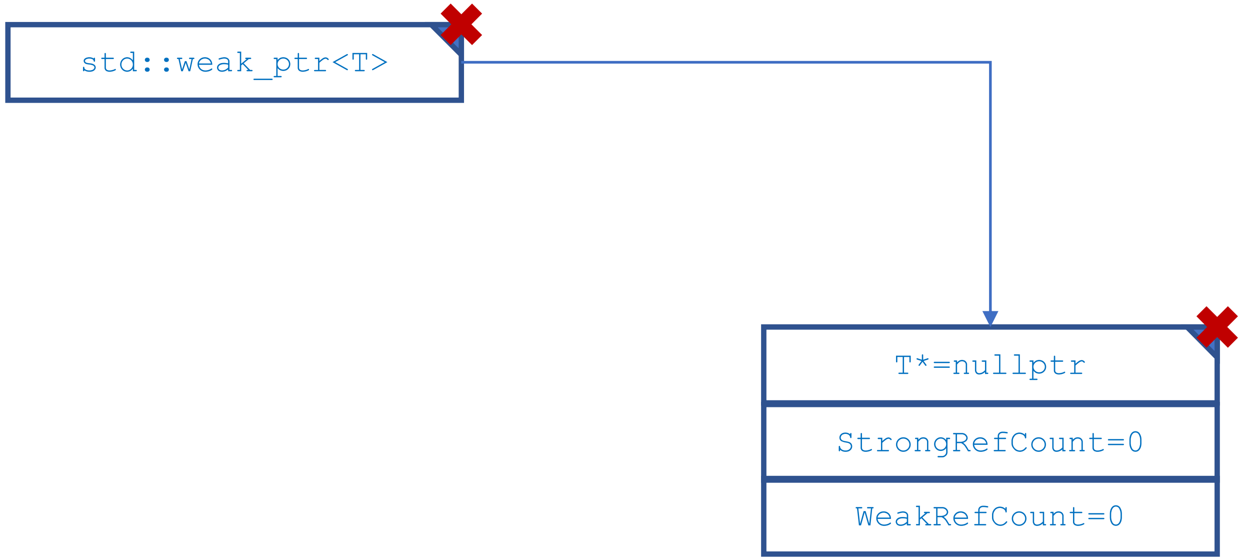


std::weak_ptr<T>
std::shared_ptr<T>
std::shared_ptr<T>
T*
StrongRefCount=1
WeakRefCount=1+1
T

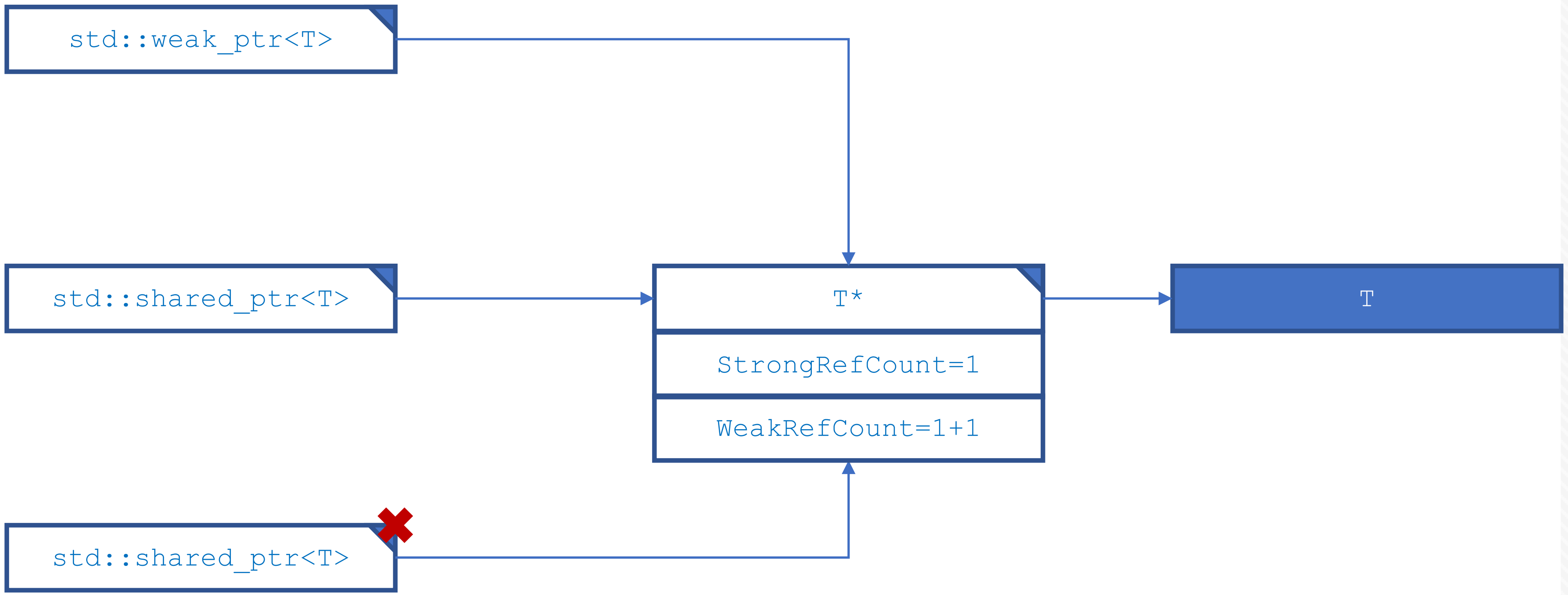
std::weak_ptr


std::weak_ptr<T>
std::shared_ptr<T>
T*
StrongRefCount=0
WeakRefCount=1
T

std::weak_ptr

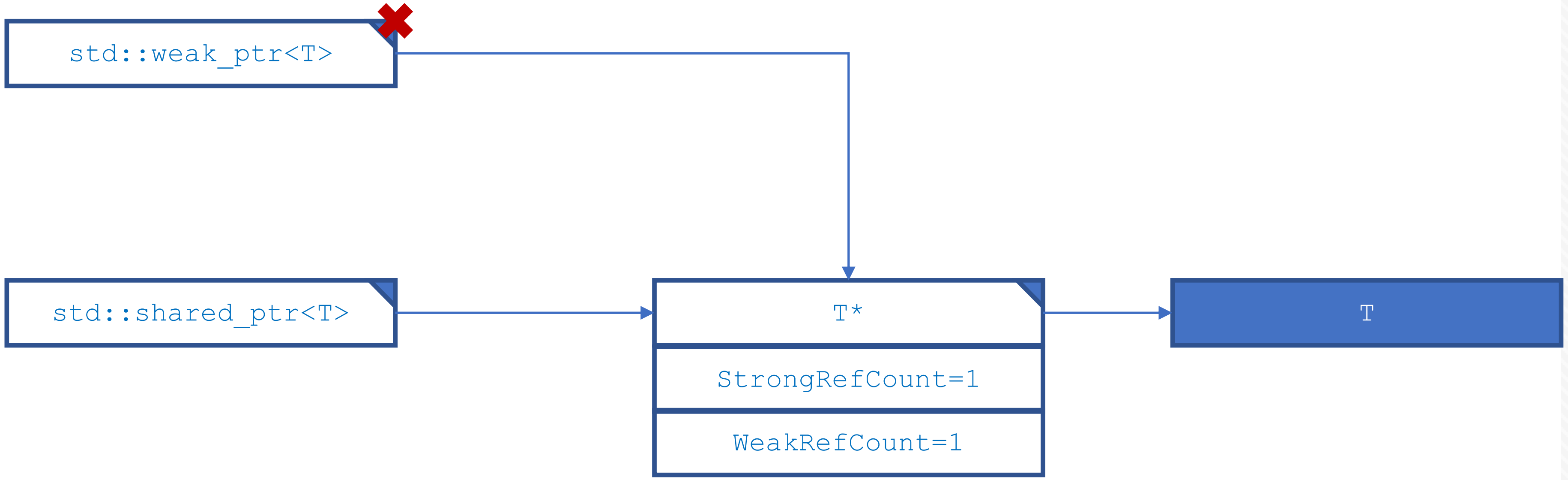


std::weak_ptr



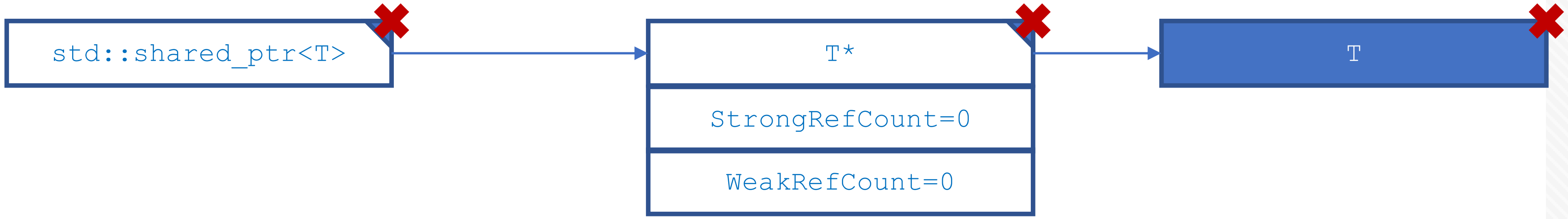
std::weak_ptr<T>
std::shared_ptr<T>
T*
StrongRefCount=1
WeakRefCount=1+1
T
std::shared_ptr<T>

std::weak_ptr



std::weak_ptr<T>
std::shared_ptr<T>
T*
StrongRefCount=1
WeakRefCount=1
T

std::weak_ptr



std::weak_ptr

```
#include <memory>
#include <iostream>

struct MyClass {
    void print() const { std::cout << "Hello world!" << std::endl; }
};

std::shared_ptr<MyClass> create() { return std::make_shared<MyClass>(/* c-tor params */); }

void print1(MyClass& obj) { obj.print(); }

template <typename T>
void print2(T&& obj_ptr) { std::forward<T>(obj_ptr)->print(); }

void print3(std::weak_ptr<MyClass> w) {
    if (std::shared_ptr<MyClass> obj = w.lock())  // Locking: weak to shared pointer.
        obj->print();
}

int main() {
    std::shared_ptr<MyClass> obj = create();
    print1(*obj);              // Dereference
    print2(obj.get());         // Pointer
    print2(obj);               // Smart pointer
    print3(obj);               // Copied as a weak pointer
}
```

`std::weak_ptr`

- Creating
  - Using a constructor
    ```
    std::weak_ptr<T>{s1}
    std::weak_ptr<T>{w1}
    ```
  - Sharing address without ownership by copy semantics (including passing by-value)
    ```
    std::weak_ptr<T> w1 = s1;
    std::weak_ptr<T> w2 = w1;
    ```

- Using
  - Offers `lock()` which produces `shared_ptr` with `nullptr` on failure.
    ```
    if (std::shared_ptr<T> s1 = w1.lock()) {}
    ```
  - Offers `expired()` for checking if related `shared_ptr` are destroyed.
  - Offers `use_count()` exposing its reference counting mechanism.
  - Does not offer pointer-like syntax; does not behave like a pointer.

Exceptions

<u>Keywords</u>

`try` – indicates the level of the stack where exceptions are handled

`catch (T e)` – represents statements for handling exceptions of type `T`

`catch (...)` – represents statements for handling exceptions of any type

`throw obj;` – uses `obj` as a description of an exceptional situation and begins the **stack unwinding** process

`throw;` – in `catch` resumes the stack unwinding process for the current description object (rethrow).

Exceptions

Example

```
try
{
    doBefore();
    doSomething();
    doLater();
}
catch (const std::exception& e)
{
    std::cerr << e.what() << std::endl;
    throw;
}
catch (...)
{
    std::cerr << "Oops!" << std::endl;
}
```

```
void doSomething()
{
    if (isTrue())
    {
        throw std::runtime_error{
            "Something went wrong!"
        };
    }
    // Do some work
}
```

Exceptions

<u>Stack unwinding</u>

- *Non-class automatic object* – pop from the stack.
- *Class automatic object* – call d-tor (if any) and pop from the stack.
- *Function call stack frame* – get the return address, jump back to that address, continue unwinding.
- Level indicated by `try` – match `catch` clauses:
  - On a match, execute the clause and continue without unwinding, unless an exception is rethrown.
  - Without a match, continue unwinding.
- Bottom of the call stack – call `std::terminate();`

Exceptions

The good

- We have a standardized way of handing exceptional situations, catching and rethrowing across functions.
- If something went wrong, jump over subsequent operations directly to the handling code.
- Pass an exception object, instead of reserved values, special result objects, or additional output parameters.

Exceptions

<u>The bad</u>

- Pointers as non-class objects are popped without clean-up.
- Resources tracked by non-class objects are popped without clean-up (i.e. Windows GDI).
- Class objects holding resources must implement clean-up through a destructor (the Rule of N).

Exception safety guarantees

The following generally accepted levels are used for documenting guarantees functions give about exceptions:

- No exception guarantee.
- Basic exception guarantee.
- Strong exception guarantee.
- Nothrow/nofail exception guarantee.

Exception safety guarantees

No exception guarantee

- The code may throw exceptions, leak resources, and objects may end up in an invalid state in exceptional situations.

Exception safety guarantees

Basic exception guarantee

- The code may throw exceptions, but it guarantees no resource leaks in successful execution and exceptional situations.
- The objects may end up in an invalid "*business state*" (for example, enter a special failure state), but their encapsulated "*logic state*" always remains valid (for example there are no dangling pointers; they can be still used, assigned or deleted).
- Most functions and standard libraries offer this guarantee.

Exception safety guarantees

Strong exception guarantee

- Commit or rollback.
- Full basic guarantee (i.e. no leaks) with no side effects or committed data loss in case of operation failure.
- May require more complex implementation
- May require extra computational power or memory to revert incomplete changes.

Exception safety guarantees

Nothrow/nofail exception guarantee

- Full strong guarantee with failure transparency:
  - Nothrow (exceptions are caught internally and never rethrown) is expected from functions called during stack unwinding.
  - Nofail (operations always succeed) is expected from swaps (i.e. `std::swap`), move constructors, and move assignment operators.
- Possible only for some algorithms.
- Functions with the guarantee can be in C++ marked using a `noexcept` specifier (technically, they can `throw` exceptions or call other functions that throw, but this behaviour ends with an immediate `std::terminate()` call).